# 7th International Conference on Formal Structures for Computation and Deduction

**FSCD 2022, August 2–5, 2022, Haifa, Israel**

Edited by

Amy P. Felty

*Editors*

**Amy P. Felty**
University of Ottawa, Canada
afelty@uottawa.ca

*ACM Classification 2012*
Theory of computation → Models of computation; Theory of computation → Logic; Theory of computation → Semantics and reasoning; Theory of computation → Formal languages and automata theory; Software and its engineering → Formal software verification; Software and its engineering → Formal language definitions; Software and its engineering → Software organization and properties; Security and privacy → Formal methods and theory of security

**ISBN 978-3-95977-233-4**

*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/978-3-95977-233-4.

*Publication date*
June, 2022

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

*License*
This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): https://creativecommons.org/licenses/by/4.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.FSCD.2022.0

## LIPIcs – Leibniz International Proceedings in Informatics

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

*Editorial Board*

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

**ISSN 1868-8969**

**https://www.dagstuhl.de/lipics**

# Contents

Preface
*Amy P. Felty* ........ 0:ix

Committees
........ 0:xi

External Reviewers
........ 0:xiii

Authors
........ 0:xv

## Invited Talks

Cutting a Proof into Bite-Sized Chunks: Incrementally proving termination in higher-order term rewriting
*Cynthia Kop* ........ 1:1–1:17

A Methodology for Designing Proof Search Calculi for Non-Classical Logics
*Alwen Tiu* ........ 2:1–2:4

## Regular Papers

A Fibrational Tale of Operational Logical Relations
*Francesco Dagnino and Francesco Gavazzo* ........ 3:1–3:21

On Quantitative Algebraic Higher-Order Theories
*Ugo Dal Lago, Furio Honsell, Marina Lenisa, and Paolo Pistone* ........ 4:1–4:18

Sheaf Semantics of Termination-Insensitive Noninterference
*Jonathan Sterling and Robert Harper* ........ 5:1–5:19

Combined Hierarchical Matching: the Regular Case
*Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen* ........ 6:1–6:22

Nominal Anti-Unification with Atom-Variables
*Manfred Schmidt-Schaußand Daniele Nantes-Sobrinho* ........ 7:1–7:22

A Certified Algorithm for AC-Unification
*Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes Sobrinho* ........ 8:1–8:21

An Analysis of Tennenbaum's Theorem in Constructive Type Theory
*Marc Hermes and Dominik Kirst* ........ 9:1–9:19

Constructing Unprejudiced Extensional Type Theories with Choices via Modalities
*Liron Cohen and Vincent Rahli* ........ 10:1–10:23

Division by Two, in Homotopy Type Theory
*Samuel Mimram and Émile Oleon* ........ 11:1–11:17

7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Type-Based Termination for Futures
*Siva Somayyajula and Frank Pfenning* ........ 12:1–12:21

Addition and Differentiation of ZX-Diagrams
*Emmanuel Jeandel, Simon Perdrix, and Margarita Veshchezerova* ........ 13:1–13:19

Restricting Tree Grammars with Term Rewriting
*Jan Bessai, Lukasz Czajka, Felix Laarmann, and Jakob Rehof* ........ 14:1–14:19

On Lookaheads in Regular Expressions with Backreferences
*Nariyoshi Chida and Tachio Terauchi* ........ 15:1–15:18

Certified Decision Procedures for Two-Counter Machines
*Andrej Dudenhefner* ........ 16:1–16:18

Strategies for Asymptotic Normalization
*Claudia Faggian and Giulio Guerrieri* ........ 17:1–17:24

Solvability for Generalized Applications
*Delia Kesner and Loïc Peyrot* ........ 18:1–18:22

Normalization Without Syntax
*Willem B. Heijltjes, Dominic J. D. Hughes, and Lutz Straßburger* ........ 19:1–19:19

Decision Problems for Linear Logic with Least and Greatest Fixed Points
*Anupam Das, Abhishek De, and Alexis Saurin* ........ 20:1–20:20

Linear Lambda-Calculus is Linear
*Alejandro Díaz-Caro and Gilles Dowek* ........ 21:1–21:17

A Graphical Proof Theory of Logical Time
*Matteo Acclavio, Ross Horne, Sjouke Mauw, and Lutz Straßburger* ........ 22:1–22:25

A Stratified Approach to Löb Induction
*Daniel Gratzer and Lars Birkedal* ........ 23:1–23:22

Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity
*Frédéric Blanqui* ........ 24:1–24:14

Adequate and Computational Encodings in the Logical Framework Dedukti
*Thiago Felicissimo* ........ 25:1–25:18

mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity
*Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller* ........ 26:1–26:23

Polynomial Termination Over ℕ Is Undecidable
*Fabian Mitterwallner and Aart Middeldorp* ........ 27:1–27:17

Compositional Confluence Criteria
*Kiraku Shintani and Nao Hirokawa* ........ 28:1–28:19

Rewriting for Monoidal Closed Categories
*Mario Alvarez-Picallo, Dan Ghica, David Sprunger, and Fabio Zanasi* ........ 29:1–29:20

Stateful Structural Operational Semantics
*Sergey Goncharov, Stefan Milius, Lutz Schröder, Stelios Tsampas, and Henning Urbat* ........ 30:1–30:19

A Combinatorial Approach to Higher-Order Structure for Polynomial Functors
*Marcelo Fiore, Zeinab Galal, and Hugo Paquet* ........ 31:1–31:19

Galois Connecting Call-by-Value and Call-by-Name
*Dylan McDermott and Alan Mycroft* ........ 32:1–32:19

# Preface

This volume contains the proceedings of the 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022), which was held August 2–5, 2022 in Haifa, Israel, as part of the Federated Logic Conference (FLoC).

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, as well as semantics and verification in new and challenging areas.

The FSCD program featured two invited talks given by Cynthia Kop (Radboud University Nijmegen) and Alwen Tiu (The Australian National University). In addition, a FLoC Keynote talk was given by Catuscia Palamidessi (Inria Saclay and LIX) and a FLoC Plenary talk was given by Orna Kupferman (Hebrew University). Participants also had the opportunity to participate in a special session jointly organized with the Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP), celebrating Frank Pfenning's contributions on the occasion of his (belated) 60th birthday.

The program committee consisted of 33 members from 12 countries. Each submitted paper was reviewed by at least three PC members with the help of 45 external reviewers. The reviewing process, which included a rebuttal phase, took place over eight weeks. A total of 31 regular research papers were accepted for publication and are included in these proceedings. The Program Committee awarded the FSCD 2022 Best Paper Award by Junior Researchers to Marc Hermes and Dominik Kirst for their paper "An Analysis of Tennenbaum's Theorem in Constructive Type Theory."

In addition to the main program, 8 FSCD-associated workshops were held before the conference:

- IFIP-WG1.6: Annual Meeting of the IFIP Working Group 1.6 on Term Rewriting
- HoTT/UF: 7th Workshop on Homotopy Type Theory/Univalent Foundations
- IWC: 11th International Workshop on Confluence
- LFMTP: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice
- Linearity-TLLA: 3rd Joint International Workshop on Linearity in Logic and Computer science and its Applications
- TERMGRAPH: 12th International Workshop on Computing with Terms and Graphs
- WiL: 6th Workshop on Women in Logic
- WPTE: 9th International Workshop on Rewriting Techniques for Program Transformations and Evaluation

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 and Michael Didas for their prompt replies to any questions regarding the production of these proceedings.

Many people have contributed to the success of FSCD 2022. On behalf of the Program Committee, I thank the authors of submitted papers for considering FSCD as a venue for their work. The Program Committee and the external reviewers deserve special thanks for their careful review and evaluation of the submitted papers. We also thank all invited speakers for

enriching both FSCD and FLoC with their talks. In addition, we acknowledge the important contributions of the workshop organizers who were enthusiastic about co-locating with FSCD; their efforts helped to ensure that workshops continue to be an important element of FSCD. Nachum Dershowitz, the Conference Chair for FSCD 2022, deserves a warm thanks for the organization of FSCD 2022 and for producing the web site. 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. Thanks also to the FLoC Organizing Committee, Program Committee, and Steering Committee for their tireless work in coordinating the overall effort of this large event. Finally, I thank all participants of the conference for creating a lively and interesting event, especially after two consecutive years of virtual-only events.

FSCD 2022 was held in-cooperation with ACM SIGLOG and ACM SIGPLAN. As part of FLoC, it was also supported by AWS, Meta, Intel, Google, Synopsis, Cadence, DLVSystem, Veridise, Technion, and The Henry and Marilyn Taub Faculty of Computer Science at Technion.

Amy Felty
Program Chair of FSCD 2022

# Committees

## PROGRAM COMMITTEE

| | |
|---|---|
| Amal Ahmed | Northeastern University |
| Thorsten Altenkirch | Nottingham University |
| Takahito Aoto | Niigata University |
| Kazuyuki Asada | Tohoku University |
| Franz Baader | TU Dresden |
| James Cheney | University of Edinburgh |
| Agata Ciabattoni | Vienna University of Technology |
| Horatiu Cirstea | Loria |
| Nachum Dershowitz | Tel Aviv University |
| Gilles Dowek | Inria & ENS Paris-Saclay |
| Amy Felty (Chair) | University of Ottawa |
| Carsten Fuhs | Birkbeck, University of London |
| Hugo Herbelin | Inria & University of Paris |
| Patricia Johann | Appalachian State University |
| Daniel Licata | Wesleyan University |
| Salvador Lucas | University Politècnica de València |
| Christopher Lynch | Clarkson University |
| Ralph Matthes | IRIT, CNRS, TU Toulouse |
| Paul-André Melliès | CNRS, University of Paris |
| Alexandre Miquel | Universidad de la República |
| Georg Moser | University of Innsbruck |
| Daniele Nantes | University of Brasília |
| Vivek Nigam | Huawei ERC & UFPB |
| Carlos Olarte | UFRN |
| Valeria de Paiva | Topos Institute |
| Giselle Reis | CMU Qatar |
| Masahiko Sakai | Nagoya University |
| Renate Schmidt | University of Manchester |
| Martina Seidl | Johannes Kepler University |
| Sam Staton | University of Oxford |
| Christine Tasson | Sorbonne University |
| Benoît Valiron | LRI & University of Paris |
| Stephanie Weirich | University of Pennsylvania |

## CONFERENCE CHAIR

| | |
|---|---|
| Nachum Dershowitz | Tel Aviv University |

## PUBLICITY CHAIR

| | |
|---|---|
| Carsten Fuhs | Birkbeck, University of London |

7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## STEERING COMMITTEE

| | |
|---|---|
| Zena M. Ariola | University of Oregon |
| Alejandro Díaz-Caro | Universidad Nacional de Quilmes & ICC (UBA/CONICET) |
| Carsten Fuhs | Birkbeck, University of London |
| Stefano Guerrini | CNRS, Université Sorbonne Paris Nord |
| Herman Geuvers (Chair) | Radboud University & Eindhoven University of Technology |
| Silvia Ghilezan | University of Novi Sad & Mathematical Institute SASA |
| Delia Kesner | Université de Paris |
| Naoki Kobayashi | The University of Tokyo |
| Luigi Liquori | Inria |
| Damiano Mazza | CNRS, Université Sorbonne Paris Nord |
| Jakob Rehof | TU Dortmund |
| Jamie Vicary | University of Cambridge |

# External Reviewers

Nathanael Arkor
Clement Aubert
Martin Avanzini
Mauricio Ayala-Rincón
Marco Benini
Martin Berglund
Eduardo Bonelli
James Brotherston
David Cerna
Jules Chouquet
Bruno Courcelle
Anupam Das
Daniel Dougherty
Harley Eades III
Marcelo Fiore
Naohiko Hoshino
Florent Jacqumard
Junyoung Jang
Vincent Jugé
Kentaro Kikuchi
Daisuke Kimura
Temur Kutsia
Stepan Kuznetsov
Jean-Jacques Levy
Sophie Libkind
Bruno Lopes
Giulio Manzonetto
Radu Mardare
Sean Moss
Keisuke Nakano
Romain Péchoux
Jorge A. Pérez
Florian Rabe
Sven Schneider
Stefan Schupp
Jonas Schöpf
Ashish Tiwari
Davide Trotta
Takeshi Tsukada
John van de Wetering
Vincent Van Oostrom
Femke Van Raamsdonk
Daniel Ventura
Uwe Waldmann
Florian Zuleger

7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# List of Authors


Matteo Acclavio (22)
Department of Computer Science,
University of Luxembourg, Luxembourg

Mario Alvarez-Picallo (29)
Programming Languages Laboratory,
Huawei Research Centre, UK

Clément Aubert (26)
School of Computer and Cyber Sciences,
Augusta University, GA, USA

Mauricio Ayala-Rincón (8)
Departments of Computer Science and
Mathematics, University of Brasília, Brazil

Jan Bessai (14)
TU Dortmund, Germany

Lars Birkedal (23)
Aarhus University, Denmark

Frédéric Blanqui (24)
Université Paris-Saclay, INRIA, ENS
Paris-Saclay, CNRS, Laboratoire Méthodes
Formelles, 4 avenue des Sciences 91190
Gif-sur-Yvette, France

Nariyoshi Chida (15)
NTT Corporation, Tokyo, Japan;
Waseda University, Tokyo, Japan

Liron Cohen (10)
Ben-Gurion University of the Negev,
Beer-Sheva, Israel

Lukasz Czajka (14)
TU Dortmund, Germany

Francesco Dagnino (3)
University of Genova, Italy

Ugo Dal Lago (4)
Department of Computer Science and
Engineering, University of Bologna, Italy

Anupam Das (20)
University of Birmingham, UK

Abhishek De (20)
IRIF, CNRS, Université Paris Cité & INRIA,
France

Gilles Dowek (21)
Inria, Paris, France;
ENS Paris-Saclay, France

Andrej Dudenhefner (16)
TU Dortmund, Germany

Alejandro Díaz-Caro (21)
Departamento de Ciencia y Tecnología,
Universidad Nacional de Quilmes, Bernal,
Buenos Aires, Argentina;
Instituto de Ciencias de la Computación,
CONICET / Universidad de Buenos Aires,
Buenos Aires, Argentina

Serdar Erbatur (6)
University of Texas at Dallas, TX, USA

Claudia Faggian (17)
IRIF, CNRS, Université de Paris Cité,
F-75013 Paris, France

Thiago Felicissimo (25)
Université Paris-Saclay, INRIA project
Deducteam, Laboratoire de Méthodes Formelles,
ENS Paris-Saclay, 91190, France

Maribel Fernández (8)
Department of Informatics,
King's College London, UK

Marcelo Fiore (31)
University of Cambridge, UK

Zeinab Galal (31)
University of Leeds, UK

Francesco Gavazzo (3)
University of Bologna, Italy

Dan Ghica (29)
Department of Computer Science,
University of Birmingham, UK;
Programming Languages Laboratory,
Huawei Research Centre, Reading, UK

Sergey Goncharov (30)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Daniel Gratzer (23)
Aarhus University, Denmark

Giulio Guerrieri (17)
Huawei Research,
Edinburgh Research Centre, UK

Robert Harper (5)
Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA, USA

Willem B. Heijltjes (19)
Department of Computer Science,
University of Bath, UK

Marc Hermes (9)
Department of Mathematics, Universität des Saarlandes, Saarbrücken, Germany

Nao Hirokawa (28)
Japan Advanced Institute of Science and Technology, Ishikawa, Japan

Furio Honsell (4)
Department of Mathematical Sciences, Informatics and Physics, University of Udine, Italy

Ross Horne (22)
Department of Computer Science,
University of Luxembourg, Luxembourg

Dominic J. D. Hughes (19)
Logic Group, University of California Berkeley, CA, USA

Emmanuel Jeandel (13)
LORIA, CNRS, Université de Lorraine,
Inria Mocqua, Nancy, France

Delia Kesner (18)
Université de Paris, CNRS, IRIF, France;
Institut Universitaire de France, France

Dominik Kirst (9)
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

Cynthia Kop (1)
Department of Software Science,
Radboud University Nijmegen, The Netherlands

Felix Laarmann (14)
TU Dortmund, Germany

Marina Lenisa (4)
Department of Mathematical Sciences, Informatics and Physics, University of Udine, Italy

Andrew M. Marshall (6)
University of Mary Washington,
Fredericksburg, VA, USA

Sjouke Mauw (22)
Department of Computer Science,
University of Luxembourg, Luxembourg

Dylan McDermott (32)
Reykjavik University, Iceland

Aart Middeldorp (27)
Department of Computer Science,
Universiät Innsbruck, Austria;
Future Value Creation Research Center,
Nagoya University, Japan

Stefan Milius (30)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Samuel Mimram (11)
École polytechnique, Palaiseau, France

Fabian Mitterwallner (27)
Department of Computer Science,
Universität Innsbruck, Austria

Alan Mycroft (32)
University of Cambridge, UK

Daniele Nantes-Sobrinho (7)
Department of Computing,
Imperial College London, UK;
Department of Mathematics,
University of Brasília, Brazil

Émile Oleon (11)
École polytechnique, Palaiseau, France

Hugo Paquet (31)
University of Oxford, UK

Simon Perdrix (13)
LORIA, CNRS, Université de Lorraine,
Inria Mocqua, Nancy, France

Loïc Peyrot (18)
Université de Paris, CNRS, IRIF, France

Frank Pfenning (12)
Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA, USA

Paolo Pistone (4)
Department of Computer Science and Engineering, University of Bologna, Italy

Vincent Rahli (10)
University of Birmingham, UK

Jakob Rehof (14)
TU Dortmund, Germany

Christophe Ringeissen (6)
Université de Lorraine, CNRS, Inria, LORIA,
F-54000 Nancy, France

Thomas Rubiano (26)
LIPN – UMR 7030 Université Sorbonne Paris Nord, France

Neea Rusch (26)
School of Computer and Cyber Sciences,
Augusta University, GA, USA

Alexis Saurin (20)
IRIF, CNRS, Université Paris Cité & INRIA,
France

Manfred Schmidt-Schauß (7)
Goethe Universität, Frankfurt am Main,
Germany

Lutz Schröder (30)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Thomas Seiller (26)
LIPN – UMR 7030 Université Sorbonne Paris
Nord, France;
CNRS, Paris, France

Kiraku Shintani (28)
Japan Advanced Institute of Science and
Technology, Ishikawa, Japan

Gabriel Ferreira Silva (8)
Department of Computer Science,
University of Brasília, Brazil

Daniele Nantes Sobrinho (8)
Department of Computing,
Imperial College London, UK;
Department of Mathematics,
University of Brasília, Brazil

Siva Somayyajula (12)
Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA, USA

David Sprunger (29)
Department of Computer Science,
University of Birmingham, UK

Jonathan Sterling (5)
Department of Computer Science,
Aarhus University, Denmark

Lutz Straßburger (19, 22)
Equipe Partout, Inria Saclay, Palaiseau, France

Tachio Terauchi (15)
Waseda University, Tokyo, Japan

Alwen Tiu (2)
School of Computing, The Australian National
University, Canberra, Australia

Stelios Tsampas (30)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Henning Urbat (30)
Friedrich-Alexander-Universität
Erlangen-Nürnberg, Germany

Margarita Veshchezerova (13)
LORIA, CNRS, Université de Lorraine,
Inria Mocqua, Nancy, France;
EDF R&D, France

Fabio Zanasi (29)
Department of Computer Science,
University College London, UK

# Cutting a Proof into Bite-Sized Chunks

## Incrementally proving termination in higher-order term rewriting

**Cynthia Kop**
Department of Software Science, Radboud University Nijmegen, The Netherlands

**Abstract**

This paper discusses a number of methods to prove termination of higher-order term rewriting systems, with a particular focus on *large* systems. In first-order term rewriting, the dependency pair framework can be used to split up a large termination problem into multiple (much) smaller components that can be solved individually. This is important because a large problem may take exponentially longer to solve in one go than solving each of its components.

Unfortunately, while there are higher-order versions of several of these methods, they often fail to simplify a problem enough. Here, we will explore some of these techniques and their limitations, and discuss what else can be done to incrementally build a termination proof for higher-order systems.

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

**Keywords and phrases** Termination, Modularity, Higher-order term rewriting, Dependency Pairs, Algebra Interpretations

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.1

**Category** Invited Talk

**Funding** The author is 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

In the last few decades, the term rewriting community has developed a wide scala of techniques to prove termination of term rewriting systems. A variety of automatic termination analysis tools compete against each other in the annual termination competition [23], using hundreds of different techniques. Many of these techniques can be adapted to other forms of rewriting (e.g., context-sensitive, conditional), or real-world programming languages.

*Higher-order* term rewriting systems in particular are very close to functional programming languages, and ideas developed in one are likely to extend to the other. However, realistic (functional) programs often have thousands of lines. Many termination techniques are ill-equipped for this. For example, naively finding a suitable polynomial interpretation or path ordering is exponential in the size of the TRS.

Ideally, we would like to split up a large TRS into many small parts; prove termination of each, and conclude termination of the whole. Unfortunately, this is in general impossible, as termination is not modular [21]. Instead, we may look to different properties than termination. The *dependency pair framework* [12] is a de facto standard for termination proofs in first-order term rewriting, which combines various techniques to do exactly this: a termination problem is translated into one or more *DP problems*, which are gradually simplified, split up, and eventually closed, without ever having to apply an exponential technique on all rules at once.

The DP framework has been extended to higher-order rewriting [1, 11, 16, 18]. However, some methods in the framework adapt poorly to higher-order rules; in particular *usable rules* – an important technique to remove large numbers of rules from a DP problem – are likely to fail. Hence, even with dependency pairs, we often need to find an ordering for thousands of rules at once. Hence, it seems important to develop incremental ways to find an ordering.

© Cynthia Kop;
licensed under Creative Commons License CC-BY 4.0

In this paper, I will highlight how higher-order dependency pairs can be used to cut termination proofs into (potentially many) smaller proof obligations, and where this approach is weak. In addition, I will sketch a way to incrementally build a term ordering using *tuple interpretations* [17], a recently developed methodology based on algebra interpretations [10, 20] which was designed for *complexity analysis*, but also proves very powerful for termination.

**Contribution.** This paper introduces usable rules with respect to an argument filtering for higher-order term rewriting, and lifts the arity restrictions in weakly monotonic interpretations [10]. However, the purpose of this paper is not to introduce new theory, but rather to explain how known techniques can be applied to build up a higher-order termination proof in many small steps. Hence, we will focus on a simple format that allows for an easy presentation.

**Related work.** Aside from various definitions of dependency pairs, the most relevant related work is a recent approach by Hamana [13] which aims to split up a TRS into two parts: one which should be proved terminating when combined with some simple additional rules, the other ordered by a specific technique. This is discussed a bit further in Section 4.

## 2 Preliminaries

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of similar but not fully compatible systems aiming to combine term rewriting and typed $\lambda$-calculi. Since this paper aims to explain *ideas* rather than provide technical detail, we will use a formalism that allows for a simple presentation: simply-typed $\lambda$-calculus with base-type rules and plain matching. The ideas extend to other forms of higher-order rewriting, but most definitions (e.g., dependency pairs) need more cases there.

Given a set $\mathbb{S}$ of *sorts*, the set $\mathbb{ST}$ of *simple types* is given by: (a) $\mathbb{S} \subseteq \mathbb{ST}$ and (b) if $\sigma, \tau \in \mathbb{ST}$ then $\sigma \Rightarrow \tau \in \mathbb{ST}$. Types are denoted $\sigma, \tau, \rho$ and sorts $\iota, \kappa$. We let $\Rightarrow$ be right-associative. Hence, all types have a unique representation in the form $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$.

We assume given disjoint sets $\mathcal{F}$ of typed function symbols, notation $(\mathtt{f} :: \sigma) \in \mathcal{F}$, and $\mathcal{V}$ of typed variables, notation $(x :: \sigma) \in \mathcal{V}$; there should be countably many variables of each type. *Terms* are expressions $s$ where $s :: \sigma$ can be inductively derived for some $\sigma$ by: (a) $a :: \sigma$ if $(a :: \sigma) \in \mathcal{F} \cup \mathcal{V}$; (b) $s\ t :: \tau$ if $s :: \sigma \Rightarrow \tau$ and $t :: \sigma$; (c) $\lambda x.s :: \sigma \Rightarrow \tau$ if $(x :: \sigma) \in \mathcal{V}$ and $s :: \tau$. The $\lambda$ binds variables as in the $\lambda$-calculus; unbound variables are called *free* and $\mathcal{FV}(s)$ is the set of variables occurring unbound in $s$. A term $s$ is called *closed* if $\mathcal{FV}(s) = \emptyset$. Term equality is modulo $\alpha$-conversion. Application is left-associative. A term $s$ *has type* $\sigma$ if $s :: \sigma$; it *has base type* if $\sigma \in \mathbb{S}$. The *head symbol* of a term $\mathtt{f}\ s_1 \cdots s_n$ is $\mathtt{f}$.

A term $s$ has a *maximally applied subterm* $t$, notation $s \trianglerighteq t$, if either $s = t$, or $s \triangleright t$, where $s \triangleright t$ if (a) $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{F} \cup \mathcal{V}$ and some $s_i \trianglerighteq t$; or (b) $s = (\lambda x.u)\ s_1 \cdots s_n$ (with $n \geq 0$) and some $s_i \trianglerighteq t$ or $u \trianglerighteq t$. Note that *not* $s\ t \trianglerighteq s$. A *pattern* is a term $s$ such that whenever $s \trianglerighteq t\ s_1 \cdots s_n$ with $n > 0$ then $t$ is not an abstraction or an element of $\mathcal{FV}(s)$.

A substitution is a type-preserving mapping from variables to terms. The *domain* of a substitution $\gamma$ is the set $\{x \in \mathcal{V} \mid \gamma(x) \neq x\}$. Substitution does not capture bound variables; we let: (a) $x\gamma = \gamma(x)$; (b) $\mathtt{f}\gamma = \mathtt{f}$; (c) $(s\ t)\gamma = (s\gamma)\ (t\gamma)$ and (d) $(\lambda x.s)\gamma = \lambda x.(s\gamma)$ if $\gamma(x) = x$ and there is no $y$ such that $x \in \mathcal{FV}(\gamma(y))$; this is always defined by $\alpha$-conversion.

A relation $\to$ on terms is *monotonic* if $s \to t$ implies $\lambda x.s \to \lambda x.t$ and $u\ s \to u\ t$ and $s\ u \to t\ u$. The relation $\to_\beta$ is the smallest monotonic relation such that $(\lambda x.s)\ t \to_\beta s[x := t]$, where $[x := t]$ is the substitution mapping $x$ to $t$. A *rewrite rule* is a pair $\ell \to r$ of a *pattern*

$\ell$ of the form $\mathtt{f}\ \ell_1 \cdots \ell_k$ and a term $r$ such that $\mathcal{FV}(r) \subseteq \mathcal{FV}(\ell)$, $\ell$ and $r$ have the same **base type**, and $r$ has no subterms of the form $(\lambda x.s)\ t_1 \cdots t_n$ with $n > 0$. Given a set of rules $\mathcal{R}$, the relation $\to_{\mathcal{R}}$ is the smallest monotonic relation on terms such that $\ell\gamma \to_{\mathcal{R}} r\gamma$ for all $\ell \to r \in \mathcal{R}$ and substitutions $\gamma$, and $\to_{\mathcal{R}}$ includes $\to_\beta$. A term $s$ is *in normal form* if there is no $t$ such that $s \to_{\mathcal{R}} t$, and it is *$\beta$-normal* if there is no $t$ such that $s \to_\beta t$. It is *terminating* if there is no infinite reduction $s \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \dots$. We say that $\to_{\mathcal{R}}$ is terminating if all terms over $\mathcal{F}, \mathcal{V}$ are terminating. The set $\mathcal{D} \subseteq \mathcal{F}$ of *defined symbols* consists of those $\mathtt{f}$ such that $\mathcal{R}$ contains a rule $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$; all other symbols are called *constructors*.

▶ Remark 1. Note that the limitation that rules have base type is not standard in the higher-order literature. We use it here to support a simpler presentation of definitions.

▶ **Example 2.** As a running example, we will use a system over sorts $\mathsf{nat}$ (natural numbers), $\mathsf{bool}$ (booleans) and $\mathsf{list}$ (lists of numbers). Let $\mathtt{0} :: \mathsf{nat}$, $\mathtt{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}$, $\top :: \mathsf{bool}$, $\bot :: \mathsf{bool}$, $\mathtt{nil} :: \mathsf{list}$, $\mathtt{cons} :: \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$; the types of other symbols can be deduced.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{map}\ F\ \mathtt{nil} & \to & \mathtt{nil} & \mathtt{map}\ F\ (\mathtt{cons}\ x\ a) & \to & \mathtt{cons}\ (F\ x)\ (\mathtt{map}\ F\ a) \\
\mathtt{fold}\ F\ x\ \mathtt{nil} & \to & x & \mathtt{fold}\ F\ x\ (\mathtt{cons}\ y\ a) & \to & \mathtt{fold}\ F\ (F\ x\ y)\ a \\
\mathtt{min}\ x\ \mathtt{0} & \to & x & \mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{min}\ x\ y \\
\mathtt{quot}\ \mathtt{0}\ (\mathtt{s}\ y) & \to & \mathtt{0} & \mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{s}\ (\mathtt{quot}\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)) \\
\mathtt{ack}\ \mathtt{0}\ y & \to & \mathtt{s}\ y & \mathtt{ack}\ (\mathtt{s}\ x)\ \mathtt{0} & \to & \mathtt{ack}\ x\ (\mathtt{s}\ \mathtt{0}) \\
\mathtt{inc}\ \mathtt{0} & \to & \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ \mathtt{0})) & \mathtt{ack}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & \to & \mathtt{ack}\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y) \\
\mathtt{exp}\ \mathtt{0}\ y & \to & y & \mathtt{exp}\ (\mathtt{s}\ x)\ y & \to & \mathtt{double}\ x\ y\ \mathtt{0} \\
\mathtt{double}\ x\ \mathtt{0}\ z & \to & \mathtt{exp}\ x\ z & \mathtt{double}\ x\ (\mathtt{s}\ y)\ z & \to & \mathtt{double}\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z)) \\
\mathtt{mkbig}\ a\ x & \to & \mathtt{map}\ (\mathtt{ack}\ x)\ a & \mathtt{mkdiv}\ a\ x & \to & \mathtt{map}\ (\lambda y.\mathtt{quot}\ y\ x)\ a \\
\mathtt{sma}\ b\ F\ \mathtt{0} & \to & \mathtt{0} & \mathtt{sma}\ \top\ F\ (\mathtt{s}\ x) & \to & \mathtt{s}\ x \\
\mathtt{sma}\ \bot\ F\ (\mathtt{s}\ x) & \to & \mathtt{sma}\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))) & & &
\end{array}
$$

In examples in this paper, we let $\mathcal{R}_{\mathtt{f}}$ denote the subset of these rules with only the rules defining $\mathtt{f}$. For example, $\mathcal{R}_{\mathtt{map}}$ refers to the top two rules, and $\mathcal{R}_{\mathtt{ack}}$ has three rules.

**Accessibility.** Given a quasi-ordering $\succeq^{\mathbb{S}}$ on $\mathbb{S}$ whose strict part $\succ^{\mathbb{S}} := \succeq^{\mathbb{S}} \setminus \preceq^{\mathbb{S}}$ is well-founded, we define, for sort $\iota$ and type $\sigma \equiv \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$, two relations: $\iota \succeq^{\mathbb{S}}_{+} \sigma$ if $\iota \succeq^{\mathbb{S}} \kappa$ and $\iota \succ^{\mathbb{S}}_{-} \sigma_i$ for all $i$, and $\iota \succ^{\mathbb{S}}_{-} \sigma$ if $\iota \succ^{\mathbb{S}} \kappa$ and $\iota \succeq^{\mathbb{S}}_{+} \sigma_i$ for all $i$. (Here, $\iota \succeq^{\mathbb{S}}_{+} \sigma$ corresponds to "$\iota$ occurs only positively in $\sigma$" in [3, 4, 6].) For $\mathtt{f} :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota$, let $Acc(\mathtt{f}) = \{i \in \{1, \dots, m\} \mid \iota \succeq^{\mathbb{S}} \sigma_i\}$ For terms $s, t$, denote $s \trianglerighteq_{\mathrm{acc}} t$ if (a) $s = t$, (b) $s = \lambda x.s'$ and $s' \trianglerighteq_{\mathrm{acc}} t$, or (c) $s = \mathtt{f}\ s_1 \cdots s_n$ and $s_i \trianglerighteq_{\mathrm{acc}} t$ for some $i \in Acc(\mathtt{f})$.

For a fixed quasi-ordering $\succeq^{\mathbb{S}}$ on sorts, a term $s :: \iota$ is *computable* iff (1) $s$ is terminating, and (2) if $s \to_{\mathcal{R}}^{*} \mathtt{f}\ s_1 \cdots s_m$ then $s_i$ is computable for all $i \in Acc(\mathtt{f})$. A term $s :: \sigma \Rightarrow \tau$ is computable iff $s\ t$ is computable for all computable terms $t :: \sigma$. Although this is not an inductive definition, computability is a definable property (see, e.g., [11]).

▶ **Example 3.** For $\mathtt{f} :: (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat}$, we have $Acc(\mathtt{f}) = \emptyset$ for any $\succeq^{\mathbb{S}}$. If $\mathsf{ord} \succ^{\mathbb{S}} \mathsf{nat}$ and $\mathtt{g} :: (\mathsf{nat} \Rightarrow \mathsf{ord}) \Rightarrow \mathsf{ord}$, then we do have $Acc(\mathtt{g}) = \{1\}$. Hence, $\mathtt{f}\ F \not\trianglerighteq_{\mathrm{acc}} F$ but $\mathtt{g}\ F \trianglerighteq_{\mathrm{acc}} F$.

**Functions and orderings.** A 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 not required that $\geq$ is the reflexive closure of $>$. If $(A_1, >_1 \geq_1)$, $\dots, (A_n, >_n \geq_n)$ are all well-founded sets, then so is $(A_1 \times \dots \times A_n, >^{\times}, \geq^{\times})$, where $\vec{a} \geq^{\times} \vec{b}$ if each $a_i \geq_i b_i$, and $\vec{a} >^{\times} \vec{b}$ if in addition $a_i >_i b_i$ for some $i$ (writing $\vec{a} := \langle a_1, \dots, a_n \rangle$).

Let $(A, >, \geq)$ and $(B, \succ, \succeq)$ be well-founded sets. $A \Longrightarrow B$ is 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 $x \in A$. Elements of $A \Longrightarrow B$ are compared pointwise: $f \sqsupset g$ if $f(x) \succ g(x)$ for all $x \in A$; and $f \sqsupseteq g$ if $f(x) \succeq g(x)$ for all $x \in A$. We say that $f \in A \Longrightarrow B$ is *weakly monotonic* if $x \geq y$ implies $f(x) \succeq g(y)$. It is *strongly monotonic* if in addition $x > y$ implies $f(x) \succ g(y)$.

## 3 Dependency pairs

The traditional way to prove termination of a TRS is to embed the rewrite relation in a well-founded ordering. This is typically done by defining a *monotonic*, *stable* ordering (stable: if $s \succ t$ then $s\gamma \succ t\gamma$ for all substitutions $\gamma$), and then showing that $\ell \succ r$ for all rules $\ell \to r$.

▶ **Example 4.** One ordering method is to map each base-type term $s$ to a natural number $[\![s]\!]$, and let $s \succ t$ if $[\![s]\!] > [\![t]\!]$. For example, for some of the symbols in Ex. 2, we may define:

$$\begin{array}{rclcrcl} [\![\mathtt{nil}]\!] & = & 0 & \quad & [\![\mathtt{map}\ F\ L]\!] & = & ([\![L]\!] + 1) * ([\![F]\!]([\![L]\!]) + 1) \\ [\![\mathtt{cons}\ H\ T]\!] & = & [\![H]\!] + [\![T]\!] + 1 & & & & \end{array}$$

Here, a term $F :: \mathsf{nat} \Rightarrow \mathsf{nat}$ is mapped to a *strongly monotonic* function in $\mathbb{N} \Rightarrow \mathbb{N}$. We can prove that $[\![\ell]\!] > [\![r]\!]$ holds for the two rules in $\mathcal{R}_{\mathtt{map}}$. Since the interpretation functions are strongly monotonic, and the method is stable by its nature, this shows termination of $\mathcal{R}_{\mathtt{map}}$.

Unfortunately, to prove termination in this way we must find an interpretation that orders all rules at the same time. In a system with thousands of rules, this may well be infeasible. We can do a bit better with *rule removal*: if $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and we have a (monotonic, stable) well-founded ordering $\succ$ and a compatible (monotonic, stable) quasi-ordering $\succeq$ on terms, and if $\ell \succ r$ for $\ell \to r \in \mathcal{R}_1$ and $\ell \succeq r$ for $\ell \to r \in \mathcal{R}_2$, then $\to_{\mathcal{R}}$ terminates if and only if $\to_{\mathcal{R}_2}$ does. Hence, having a termination proof for $\to_{\mathcal{R}_2}$ makes the termination proof for $\to_{\mathcal{R}}$ easier. However, we still have to orient all rules in $\mathcal{R}$ at once, and $\ell \succeq r$ is often not *that* much easier to show than $\ell \succ r$, partially due to the monotonicity requirement on $\succ$.

▶ **Example 5.** Commonly used orderings like the recursive path ordering and interpretations to $\mathbb{N}$ cannot handle the `quot` rules from Example 2, as the monotonicity requirement on $\succ$ essentially causes the property that, for any choice of ordering/interpretation, $\mathtt{min}\ x\ y \succeq y$; and therefore $\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ (\mathtt{s}\ x)) \succ \mathtt{s}\ (\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ (\mathtt{s}\ x)))$, contradicting well-foundedness.

The dependency pair framework addresses both these issues. There are multiple higher-order definitions of dependency pairs, with distinct advantages and downsides; here, we present a form of *static* dependency pairs, both for its ease in presentation and because the static approach allows for more modular proofs than the alternative, *dynamic* style. To use static dependency pairs, we limit interest to *accessible function passing* (AFP) rules.

▶ **Definition 6.** *A set of rules $\mathcal{R}$ is* accessible function passing *if there exists a sort ordering $\succeq^{\mathbb{S}}$ such that: for all $\mathtt{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R}$ and all $x \in \mathcal{FV}(r)$, there exists $i$ with $\ell_i \trianglerighteq_{acc} x$.*

This requirement means that higher-order variables are used in an essentially harmless way. An example of a non-AFP rule is the encoding of the untyped $\lambda$-calculus: $\mathtt{app}\ (\mathtt{lam}\ F)\ X \to F\ X$, with $\mathtt{lam} :: (\mathtt{o} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}$ and $\mathtt{app} :: \mathtt{o} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}$, where a higher-order variable is lifted out of a base-type term. There are also terminating systems which are not AFP. However, practical examples typically satisfy this requirement. For example, the rule $\mathtt{lapply}\ x\ (\mathtt{fcons}\ F\ a) \to F\ (\mathtt{lapply}\ x\ a)$ with $\mathtt{fcons} :: (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{flist} \Rightarrow \mathsf{flist}$ also lifts a higher-order variable out of a base-type term, but is AFP if we choose $\mathsf{flist} \succ^{\mathbb{S}} \mathsf{nat}$.

In this paper, we will mostly consider rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ where all higher-order variables occur as a direct argument of the left-hand side (i.e., as one of the $\ell_i$); this is the case for all rules in our running example. Such rules are AFP by letting $\succeq^{\mathbb{S}}$ equate all sorts.

▶ **Definition 7.** *For each defined symbol* $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$, *we introduce a fresh symbol* $\mathtt{f}^\sharp :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \mathtt{dp}$. *The set of* static dependency pairs *of* $\mathcal{R}$ *is given by:* $\mathtt{SDP}(\mathcal{R}) = \{\mathtt{f}^\sharp\ \ell_1 \cdots \ell_k \Rrightarrow \mathtt{g}^\sharp\ r_1 \cdots r_n\ x_{n+1} \cdots x_m \mid \mathtt{f}\ \ell_1 \cdots \ell_k \to r \in \mathcal{R} \wedge r \trianglerighteq \mathtt{g}\ r_1 \cdots r_n \wedge \mathtt{g} \in \mathcal{D} \wedge \mathtt{g}\ r_1 \cdots r_n :: \sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \wedge x_{n+1} \in \mathcal{V}_{\sigma_1}, \ldots, x_m \in \mathcal{V}_{\sigma_m}$ *are fresh variables*$\}$.

The set of static dependency pairs is obtained by taking, for each rule $\ell \to r$, all maximally applied subterms $p$ of $r$ headed by a defined symbol, if necessary applying $p$ to fresh variables to obtain a base-type term, and marking the head symbols of both $\ell$ and $p$ to indicate their special role. In the first order setting, dependency pairs trace function calls. In the (static) higher-order setting, they also trace *potential* calls: a call of function type might end up being applied to almost anything, which is represented by the fresh variables.

▶ **Example 8.** Our running example has the following dependency pairs:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A. | $\mathtt{inc}^\sharp\ \mathtt{0}$ | $\Rrightarrow$ | $\mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})$ | J. | $\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ x\ a)$ | $\Rrightarrow$ | $\mathtt{map}^\sharp\ F\ a$ |
| B. | $\mathtt{exp}^\sharp\ (\mathtt{s}\ x)\ y$ | $\Rrightarrow$ | $\mathtt{double}^\sharp\ x\ y\ \mathtt{0}$ | K. | $\mathtt{fold}^\sharp\ F\ x\ (\mathtt{cons}\ y\ a)$ | $\Rrightarrow$ | $\mathtt{fold}^\sharp\ F\ (F\ x\ y)\ a$ |
| C. | $\mathtt{min}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$ | $\Rrightarrow$ | $\mathtt{min}^\sharp\ x\ y$ | L. | $\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$ | $\Rrightarrow$ | $\mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)$ |
| D. | $\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ \mathtt{0}$ | $\Rrightarrow$ | $\mathtt{ack}^\sharp\ x\ (\mathtt{s}\ \mathtt{0})$ | M. | $\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$ | $\Rrightarrow$ | $\mathtt{min}^\sharp\ x\ y$ |
| E. | $\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$ | $\Rrightarrow$ | $\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ y$ | N. | $\mathtt{ack}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)$ | $\Rrightarrow$ | $\mathtt{ack}^\sharp\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y)$ |
| F. | $\mathtt{double}^\sharp\ x\ \mathtt{0}\ z$ | $\Rrightarrow$ | $\mathtt{exp}^\sharp\ x\ z$ | O. | $\mathtt{double}^\sharp\ x\ (\mathtt{s}\ y)\ z$ | $\Rrightarrow$ | $\mathtt{double}^\sharp\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z))$ |
| G. | $\mathtt{mkbig}^\sharp\ a\ x$ | $\Rrightarrow$ | $\mathtt{ack}^\sharp\ x\ y$ | P. | $\mathtt{mkbig}^\sharp\ a\ x$ | $\Rrightarrow$ | $\mathtt{map}^\sharp\ (\mathtt{ack}\ x)\ a$ |
| H. | $\mathtt{mkdiv}^\sharp\ a\ x$ | $\Rrightarrow$ | $\mathtt{quot}^\sharp\ y\ x$ | Q. | $\mathtt{mkdiv}^\sharp\ a\ x$ | $\Rrightarrow$ | $\mathtt{map}^\sharp\ (\lambda y.\mathtt{quot}\ y\ x)\ a$ |
| I. | $\mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x)$ | $\Rrightarrow$ | $\mathtt{quot}^\sharp\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0}))$ | R. | $\mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x)$ | $\Rrightarrow$ | $\mathtt{sma}^\sharp\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))$ |

Note that DP (G), which came from the rule $\mathtt{mkbig}\ a\ x \to \mathtt{map}\ (\mathtt{ack}\ x)\ a$, has a fresh variable $y$ in the right-hand side which does not occur on the left; this was used to flatten the subterm $\mathtt{ack}\ x$ to base type. (H) also has a variable $y$ which occurs on the right but not the left; this is because the bound variable in $\mathtt{map}\ (\lambda y.\mathtt{quot}\ y\ x)\ a$ is freed in the subterm.

Dependency pairs are used by translating non-termination to absence of infinite *chains*:

▶ **Definition 9.** *For* $\mathcal{P}$ *a set of dependency pairs, and* $\mathcal{R}$ *a set of rules, a* $(\mathcal{P},\mathcal{R})$-chain *is an infinite sequence* $[(\ell_i \Rrightarrow r_i, \gamma_i) \mid i \in \mathbb{N}]$ *such that for all* $i$: $\ell_i \Rrightarrow r_i \in \mathcal{P}$, *and* $r_i\gamma_i \to_{\mathcal{R}}^* \ell_{i+1}\gamma_{i+1}$.
*A* $(\mathcal{P},\mathcal{R})$*-chain is* computable *if each* $r_i\gamma_i$ *is computable with respect to* $\to_{\mathcal{R}}$.

Essentially, a $(\mathcal{P},\mathcal{R})$-chain represents an infinite reduction $s_1 \to_{\mathcal{P}} t_1 \to_{\mathcal{R}}^* s_2 \to_{\mathcal{P}} t_2 \to_{\mathcal{R}}^* s_3 \ldots \to_{\mathcal{P}}$, where each $s_i = \ell_i\gamma_i$ and $t_i = r_i\gamma_i$, and the steps using $\to_{\mathcal{P}}$ are at the root of $s_i$. Although chains can have various properties (e.g., being *minimal*, *computable*, *formative*), we here only consider *computability*, and only implicitly: this property – which implies that each $r_i\gamma_i$ is terminating, and that the immediate arguments of each $\ell_i\gamma_i$ are computable – is used in the (omitted) correctness proofs of Section 4. We have the following result:

▶ **Lemma 10.** *Let* $\mathcal{R}$ *be a set of accessible function passing rules (for a fixed sort ordering with* $\mathtt{dp}$ *maximal in* $\succeq^{\mathbb{S}}$*). If* $\to_{\mathcal{R}}$ *is non-terminating, then there is a computable* $(\mathtt{SDP}(\mathcal{R}),\mathcal{R})$*-chain.*

Hence, if we can prove that there is no such chain, we know the system terminates. One way of doing this is by using a well-founded ordering as before. Since the steps $s_i \to_{\mathcal{P}} t_i$ occur at the root of a term, it is not needed for $\succ$ to be monotonic. Rather, it suffices to use a *reduction pair*: a pair $(\succ, \succeq)$ that that $\succ$ is a well-founded ordering, $\succeq$ is a quasi-ordering, $\succ \cdot \succeq \subseteq \succ$, both relations are stable, $\succeq$ is monotonic, and $\to_\beta \subseteq \succeq$. We can again use interpretations to define a reduction pair. This is formally defined as follows:

▶ **Definition 11.** *We assume given, for all sorts $\iota$, a well-founded set $(\mathcal{A}_\iota, \sqsupset_\iota, \sqsupseteq_\iota)$. This definition is extended to all simple types as follows: $\mathcal{A}_{\sigma\Rightarrow\tau} = \{f \in \mathcal{A}_\sigma \Longrightarrow \mathcal{A}_\tau \mid f$ is weakly monotonic$\}$; we let $\sqsupset_{\sigma\Rightarrow\tau}$ and $\sqsupseteq_{\sigma\Rightarrow\tau}$ denote the pointwise comparisons on these functions.*

*For every $(\mathtt{f} :: \sigma) \in \mathcal{F}$, we assume given $\mathcal{J}_\mathtt{f} \in \mathcal{A}_\sigma$. For a closed term $s$ let $[\![s]\!] = [\![s]\!]_\emptyset$, where, for $\alpha$ a function mapping each $(x :: \sigma) \in \mathcal{V} \cap \mathcal{FV}(s)$ to an element of $\mathcal{A}_\sigma$, we define:*

$$[\![\mathtt{f}]\!]_\alpha = \mathcal{J}_\mathtt{f} \qquad [\![x]\!]_\alpha = \alpha(x)$$
$$[\![t\ u]\!]_\alpha = [\![t]\!]_\alpha([\![u]\!]_\alpha) \qquad [\![\lambda x.t]\!]_\alpha = d \mapsto [\![t]\!]_{\alpha[x:=d]}$$

Here, $\alpha[x := d]$ maps $x$ to $d$ and all other variables $y$ to $\alpha(y)$, and $d \mapsto [\![t]\!]_{\alpha[x:=d]}$ is the function that maps $d \in \mathcal{A}_\sigma$, to $[\![t]\!]_{\alpha[x:=d]}$. If $s :: \sigma$, this definition yields an element $[\![s]\!]_\alpha$ of $\mathcal{A}_\sigma$. We will often omit the type denotations from $\sqsupseteq$ when they are clear from context or irrelevant. We will also usually omit $\alpha$ and instead use for instance $[\![\mathtt{f}\ x]\!] = [\![x]\!] + 1$ instead of $[\![\mathtt{f}(x)]\!]_\alpha = \alpha(x) + 1$. We typically choose $[\![\cdot]\!]$ to represent a kind of size measure on terms.

▶ **Example 12.** Let $\mathcal{A}_{\mathsf{list}} = \mathbb{N}$, ordered as usual. To prove that there is no $(\mathtt{SDP}(\mathcal{R}_\mathtt{map}), \mathcal{R}_\mathtt{map})$-chain, it suffices to find an interpretation function $\mathcal{J}$ with:

$$[\![\mathtt{map}\ F\ \mathtt{nil}]\!] \geq [\![\mathtt{nil}]\!] \qquad [\![\mathtt{map}\ F\ (\mathtt{cons}\ H\ T)]\!] \geq [\![\mathtt{cons}\ (F\ H)\ (\mathtt{map}\ F\ T)]\!]$$
$$[\![\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ H\ T)]\!] > [\![\mathtt{map}^\sharp\ F\ T]\!]$$

This is easily accomplished by choosing $\mathcal{J}_\mathtt{nil} = 0$, $\mathcal{J}_\mathtt{cons}(x, y) = y + 1$, $\mathcal{J}_\mathtt{map}(F, y) = \mathcal{J}_{\mathtt{map}^\sharp}(F, y) = y$; that is, we map a term of $\mathsf{list}$ type to the length of the list. Then the above inequalities evaluate to: $0 \geq 0$, $T + 1 \geq T + 1$ and $T + 1 > T$.

Note that there is no obligation to choose $\mathcal{A}_\iota = \mathbb{N}$ for all sorts. For more complex systems than `map`, it may also be useful to for instance map sorts to the rational numbers, or to sets of terminating terms. In Section 5, we will map sorts to *tuples* of (natural) numbers.

As we have seen, dependency pairs and weakly monotonic interpretations together provide a method to prove termination. However, in contrast to the DP approach in first-order term rewriting, this is not a complete method: there are terminating systems which admit a computable chain (for example, $\mathcal{R} = \{\mathtt{f\ a} \to \mathtt{g\ f}\}$, which has a dependency pair $\mathtt{f\ a} \Rrightarrow \mathtt{f\ X}$). Hence, the method in general cannot be used for non-termination, and also has important limitations in its applicability for termination, even beyond the restriction to AFP rules.

The alternative, *dynamic* style of dependency pairs[16], does not come with applicability restrictions and does offer an if-and-only-if result. There, *collapsing* dependency pairs, of a form such as $\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ H\ T) \Rrightarrow\ F\ H$, are included, and the notion of a $(\mathcal{P}, \mathcal{R})$-chain is somewhat more complex to support this. Unfortunately, this style is much worse at enabling modular proofs. That is why this paper focuses on the static approach.

## 4 Modular proofs with dependency pairs

The dependency pair framework allows "DP problems" to be progressively modified to prove absence of chains with certain properties. We here present a very simple version of this framework, which only modifies a set $\mathcal{P}$. A more elaborate framework is discussed in [11].

We fix an AFP set $\mathcal{R}$ of rules. Let a set $\mathcal{P}$ of DPs be called *chain-free* if there is no computable $(\mathcal{P}, \mathcal{R})$-chain. Then Lemma 10 states that $\to_\mathcal{R}$ is terminating if $\mathtt{SDP}(\mathcal{R})$ is chain-free. As suggested before, sets $\mathcal{P}$ can be simplified using a reduction pair. Formally:

▶ **Lemma 13.** *A set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \mathcal{P}_1$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \mathcal{P}_2$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathcal{R}$.*

Hence, chain-freeness of $\mathcal{P}$ is reduced to chain-freeness of a smaller set. Since $\succ$ does not need to be monotonic, it is often easier to remove a dependency pair in this way than it would be to remove a rule in the original system using rule removal.

▶ **Example 14.** Let $\mathcal{R} := \mathcal{R}_{\mathtt{quot}} \cup \mathcal{R}_{\mathtt{min}} \cup \{\mathtt{inc}\ \mathtt{0} \to \mathtt{inc}\ (\mathtt{s}\ \mathtt{0})\}$. Then $\mathcal{P} := \mathtt{SDP}(\mathcal{R})$ is the set $\{(\text{A}),(\text{C}),(\text{L}),(\text{M})\}$. We choose $\mathcal{J}$ to have $[\![\mathtt{0}]\!] = 0$, $[\![\mathtt{s}\ x]\!] = [\![x]\!]+1$, $[\![\mathtt{inc}\ x]\!] = [\![\mathtt{inc}^\sharp\ x]\!] = 0$ and $[\![\mathtt{min}\ x\ y]\!] = [\![\mathtt{min}^\sharp\ x\ y]\!] = [\![\mathtt{quot}\ x\ y]\!] = [\![\mathtt{quot}^\sharp\ x\ y]\!] = [\![x]\!]$. Then $[\![\ell]\!] \geq [\![r]\!]$ for all $\ell \to r \in \mathcal{R}$, and moreover: each of (C), (L) and (M) reduces to $[\![\ell]\!] = x+1 > x = [\![r]\!]$, while for (A) we have: $[\![\ell]\!] = 0 = [\![r]\!]$. By Lemma 13, we have chain-freeness of $\mathtt{SDP}(\mathcal{R})$ (and therefore termination of $\to_{\mathcal{R}}$) if we can prove chain-freeness of $\{\mathtt{inc}^\sharp\ \mathtt{0} \Rrightarrow \mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})\}$. We avoid the problem noted in Example 5 because we only needed a *weakly* monotonic ordering.

While this is an improvement over using interpretations directly, it does nothing towards our goal: like with rule removal, in the first step we have to orient all the rules and dependency pairs in one go. Even though this is easier than before because $\succ$ does not need to be monotonic, it is still likely to be infeasible to handle thousands of rules at once.

So, let us consider an approach that does *not* need an ordering: the *splitting lemma*.

▶ **Lemma 15.** *Assume given disjoint sets of terms $A_1,\dots,A_n$, and suppose we can write $\mathcal{P} = \mathcal{P}_1 \cup \dots \cup \mathcal{P}_n \cup \mathcal{Q}_1 \cup \dots \cup \mathcal{Q}_n$ such that for all $i \in \{1,\dots,n\}$ we have:*
- *for all $\ell \Rrightarrow r \in \mathcal{P}_i \cup \mathcal{Q}_i$, and all substitutions $\gamma$: $\ell\gamma \in A_i$;*
- *for all $\ell \Rrightarrow r \in \mathcal{P}_i$, all substitutions $\gamma$ and all terms $s$ with $r\gamma \to_{\mathcal{R}}^* s$: $s \notin A_1 \cup \dots \cup A_{i-1}$;*
- *for all $\ell \Rrightarrow r \in \mathcal{Q}_i$, all substitutions $\gamma$ and all terms $s$ with $r\gamma \to_{\mathcal{R}}^* s$: $s \notin A_1 \cup \dots \cup A_i$.*

*Then $\mathcal{P}$ is chain-free if and only if $\mathcal{P}_1$, ..., $\mathcal{P}_n$ are all chain-free.*

Note that the dependency pairs in $\mathcal{Q}_1 \cup \dots \cup Q_n$ are thrown away, while the others are split over potentially many smaller sets of dependency pairs that are truly interdependent. Essentially, this lemma is a different presentation of the *DP graph processor* [2, 12, 19].

▶ **Example 16.** Let $X^{\mathtt{f}}$ denote the set $\{\mathtt{f}^\sharp\ s_1 \cdots s_m \mid (\mathtt{f} :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota) \in \mathcal{F} \wedge s_1 :: \sigma_1,\dots,s_m :: \sigma_m\}$, so the set of all base-type terms $s$ with $\mathtt{f}^\sharp$ as the head symbol.

For $\mathcal{R}$ the rules of Example 2, and $\mathcal{P} = \mathtt{SDP}(\mathcal{R})$ following Example 8, we may choose:

$$\begin{array}{lllll} A_1 := X^{\mathtt{mkbig}} & A_3 := X^{\mathtt{map}} & A_5 := X^{\mathtt{sma}} & A_7 := X^{\mathtt{min}} & A_9 := X^{\mathtt{double}} \cup X^{\mathtt{exp}} \\ A_2 := X^{\mathtt{mkdiv}} & A_4 := X^{\mathtt{fold}} & A_6 := X^{\mathtt{quot}} & A_8 := X^{\mathtt{ack}} & A_{10} := \{\mathtt{inc}^\sharp\ \mathtt{0}\} \end{array}$$

$$\begin{array}{lllll} \mathcal{P}_1 := \emptyset & \mathcal{P}_3 := \{(\text{J})\} & \mathcal{P}_5 := \{(\text{R})\} & \mathcal{P}_7 := \{(\text{C})\} & \mathcal{P}_9 := \{(\text{B}),(\text{F}),(\text{O})\} \\ \mathcal{Q}_1 := \{(\text{G}),(\text{P})\} & \mathcal{Q}_3 := \emptyset & \mathcal{Q}_5 := \{(\text{I})\} & \mathcal{Q}_7 := \emptyset & \mathcal{Q}_9 := \emptyset \\ \mathcal{P}_2 := \emptyset & \mathcal{P}_4 := \{(\text{K})\} & \mathcal{P}_6 := \{(\text{L})\} & \mathcal{P}_8 := \{(\text{D}),(\text{E}),(\text{N})\} & \mathcal{P}_{10} := \emptyset \\ \mathcal{Q}_2 := \{(\text{H}),(\text{Q})\} & \mathcal{Q}_4 := \emptyset & \mathcal{Q}_6 := \{(\text{M})\} & \mathcal{Q}_8 := \emptyset & \mathcal{Q}_{10} := \{(\text{A})\} \end{array}$$

Here, we use the property that symbols $\mathtt{f}^\sharp$ do not occur in $\mathcal{R}$, so if the right-hand of a dependency pair has the form $\mathtt{f}^\sharp\ \vec{r}$, then the same holds for each term that $(\mathtt{f}^\sharp\ \vec{r})\gamma$ reduces to. Hence, essentially, we have an ordering on the function symbols, and let $\mathcal{P}_i$ be the set of dependency pairs where both sides have a function symbol of the same weight, and $\mathcal{Q}_i$ those where the right-hand side has a smaller weight than the left. In $A_{10}$ we also consider the shape of the argument: since $\mathtt{inc}^\sharp\ (\mathtt{s}\ \mathtt{0})$ does not reduce and is not in $A_{10}$, Lemma 15 allows us to discard (A). We can also discard (G), (P), (H), (Q), (I) and (M), and reduce chain-freeness of $(\mathtt{SDP}(\mathcal{R}),\mathcal{R})$ to chain-freeness of each of $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, $\mathcal{P}_7$, $\mathcal{P}_8$ and $\mathcal{P}_9$.

Yet, this still does not really accomplish our goal: while Lemma 15 allows us to split a large set into potentially many small ones, a small set of DPs is not necessarily easy to handle. In particular, to use Lemma 13, we still need to orient all rules in $\mathcal{R}$ at once.

Fortunately, in many cases we can avoid an ordering altogether using the *subterm criterion*:

▶ **Lemma 17.** *Given a set of dependency pairs $\mathcal{P}$, and a function $\pi$ that maps each marked symbol $\mathtt{f}^\sharp :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \mathsf{dp}$ that occurs in $\mathcal{P}$ to an integer between 1 and $m$, let $\overline{\pi}(\mathtt{f}^\sharp\ s_1 \cdots s_m) := s_{\pi(\mathtt{f}^\sharp)}$. Suppose $\mathcal{P} = \mathcal{P}_= \cup \mathcal{P}_\rhd$, where $\overline{\pi}(\ell) = \overline{\pi}(r)$ for all $\ell \Rrightarrow r \in \mathcal{P}_=$ and $\overline{\pi}(\ell) \rhd \overline{\pi}(r)$ for all $\ell \Rrightarrow r \in \mathcal{P}_\rhd$. Then $\mathcal{P}$ is chain-free if and only if $\mathcal{P}_=$ is chain-free.*

The subterm criterion allows us to discard many dependency pairs without even considering $\mathcal{R}$. This is possible because the "chain-free' notion considers computable chains, so in a $(\mathcal{P}, \mathcal{R})$-chain, each $\overline{\pi}(\ell)\gamma$ and $\overline{\pi}(r)\gamma$ can be assumed to be terminating.

▶ **Example 18.** Chain-freeness of $\{(\mathrm{J})\}$ follows by $\pi(\mathtt{map}^\sharp) = 2$, since $\overline{\pi}(\mathtt{map}^\sharp\ F\ (\mathtt{cons}\ x\ a)) = \mathtt{cons}\ x\ a \rhd a = \overline{\pi}(\mathtt{map}^\sharp\ F\ a)$; we have $\mathcal{P}_= = \emptyset$ and $\mathcal{P}_\rhd = \{(\mathrm{J})\}$, and $\emptyset$ is obviously chain-free. In the same way, $\{(\mathrm{K})\}$ and $\{(\mathrm{C})\}$ are discarded (choosing $\pi(\mathtt{fold}^\sharp) = 3$ for the first, and $\pi(\mathtt{min}^\sharp) = 1$ for the second). For the set $\{(\mathrm{D}), (\mathrm{E}), (\mathrm{N})\}$, we let $\pi(\mathtt{ack}^\sharp) = 1$, and obtain chain-freeness if $\{(\mathrm{E})\}$ is chain-free, which holds by a second application of the subterm criterion, now with $\pi(\mathtt{ack}^\sharp) = 2$. For $\{(\mathrm{B}), (\mathrm{F}), (\mathrm{O})\}$, we let $\pi(\mathtt{exp}^\sharp) = \pi(\mathtt{double}^\sharp) = 1$, which allows us to discard $(\mathrm{B})$ because $\mathtt{s}\ x \rhd x$; chain-freeness of the remaining set $\{(\mathrm{F}), (\mathrm{O})\}$ follows from chain-freeness of $\{(\mathrm{O})\}$ by the splitting lemma (choosing $A_1 = X^{\mathtt{double}}$ and $A_2 = X^{\mathtt{exp}}$ as in Example 16), which follows by the subterm criterion with $\pi(\mathtt{double}^\sharp) = 2$.

Hence, following Example 16, Example 2 is terminating if $\{(\mathrm{L})\}$ and $\{(\mathrm{R})\}$ are chain-free.

The formulation and use of the subterm criterion is exactly as in the first-order case. There is a also variation of this criterion with a higher-order focus[11, Theorem 63]:

▶ **Lemma 19.** *Let $s \sqsupset t$ if $s \rhd_{acc} t$ or $t = F\ t_1 \cdots t_n$ and $s \rhd_{acc} F$ with $F \in \mathcal{V}$. $\mathcal{P}_= \cup \mathcal{P}_\rhd$ is chain-free if $\mathcal{P}_\rhd$ is chain-free, $\overline{\pi}(\ell) = \overline{\pi}(r)$ for $\ell \Rrightarrow r \in \mathcal{P}_=$ and $\overline{\pi}(\ell) \sqsupset \overline{\pi}(r)$ for $\ell \Rrightarrow r \in \mathcal{P}_\rhd$.*

So, the $\rhd$ relation in Lemma 17 is replaced by a relation that considers the type ordering and accessibility relation. This is designed particularly to handle rules like ordinal recursion: $\mathtt{rec}\ (\mathtt{lim}\ F)\ U\ X\ W \to W\ F\ (\lambda n.\mathtt{rec}\ (F\ n)\ U\ X\ W)$, which has a dependency pair $\mathtt{rec}^\sharp\ (\mathtt{lim}\ F)\ U\ X\ W \Rrightarrow \mathtt{rec}^\sharp\ (F\ n)\ U\ X\ W$ with $\mathtt{lim} :: (\mathsf{nat} \Rightarrow \mathsf{ord}) \Rightarrow \mathsf{ord}$.

The subterm criterion (whether in its basic form or the variation of Lemma 19) is a powerful technique that – in combination with the splitting lemma (Lemma 15) – might allow us to complete a termination proof in a very modular way. Yet, if any DP problems remain which cannot be further split by either lemma, we will still have to orient all the rules. To deal with this issue, we again follow the first-order DP framework and apply *usable rules*.

▶ **Definition 20** (Usable Rules)**.** *For $Q$ a set of rules or dependency pairs, let $\mathtt{rhs}(Q)$ denote the set of terms occurring as the right-hand side of some rule/DP in $Q$. For a set $T$ of terms, let $\mathtt{Use}(T, \mathcal{R})$ denote the set of those rules $\mathtt{f}\ \ell_1 \cdots \ell_k \to r$ in $\mathcal{R}$ such that:*

1. *there is a term $s \in T$ which has a (fully applied) subterm of the form $\mathtt{f}\ s_1 \cdots s_k$, or*
2. *there is a term $s \in T$ which has a subterm $x\ t_1 \cdots t_m$ with $x \in \mathcal{FV}(s)$ and $m > 0$.*

*For a set of DPs $\mathcal{P}$, we let its set $\mathsf{UR}(\mathcal{P}, \mathcal{R})$ of* usable rules *be defined as the smallest set $U \subseteq \mathcal{R}$ such that $\mathtt{Use}(\mathtt{rhs}(\mathcal{P}), \mathcal{R}) \subseteq U$ and $\mathtt{Use}(\mathtt{rhs}(U), \mathcal{R}) \subseteq U$.*

Intuitively, a rule is considered usable if we may need it to rewrite relevant instances of some right-hand side of $\mathcal{P}$. For example, when rewriting a term $\mathtt{f}\ (\mathtt{quot}\ s\ t)$, we will likely need the $\mathtt{quot}$ rules, and their use introduces occurrences of $\mathtt{min}$, which may also be relevant. However, the $\mathtt{fold}$ rules will only be used if $\mathtt{fold}$ already occurs in $s$ or $t$.

▶ **Example 21.** For our running example, $\mathsf{UR}(\{(\text{L})\ \mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \Rrightarrow \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)\}, \mathcal{R}) = \mathcal{R}_{\mathtt{min}}$, since the only defined symbol occurring in the right-hand side is $\mathtt{min}$, and the right-hand side of the two $\mathtt{min}$ rules contain no other defined symbols. Note that $\mathtt{quot}^\sharp$ is marked, and does not occur in $\mathcal{R}$, so the $\mathtt{quot}$ rules are not included. $\mathsf{UR}(\{(\text{R})\ \mathtt{sma}^\sharp\ \bot\ F\ (\mathtt{s}\ x) \Rrightarrow \mathtt{sma}^\sharp\ (F\ x)\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))\}, \mathcal{R}) = \mathcal{R}$ due to the subterm $F\ x$ of the right-hand side.

Usable rules are best used in combination with a weakly monotonic ordering. In the following, let $\mathcal{C}_\epsilon$ be a set $\{\mathtt{pair}_\iota\ x\ y \to x,\ \mathtt{pair}_\iota\ x\ y \to y \mid \iota \in \mathbb{S}\}$ for fresh symbols $\mathtt{pair}_\iota$.

▶ **Lemma 22.** *Suppose $\mathcal{R}$ is finitely branching. Then a set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \mathcal{P}_1$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \mathcal{P}_2$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_\epsilon$.*

("Finitely branching" means that for any $s$ there are only finitely many $t$ with $s \to_{\mathcal{R}} t$; this holds for instance if $\mathcal{R}$ is finite.)

The difference between Lemma 22 and Lemma 13 is that instead of orienting all rules, we only have to orient the usable rules, plus some rules of the form $\mathtt{pair}_\iota\ x_1\ x_2 \to x_i$. The latter is trivial for most commonly used orderings. The need for these additional rules is also present in the first-order case, and can be dropped when considering *innermost* termination.

▶ **Example 23.** To prove chain-freeness of $\{(\text{L})\ \mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \Rrightarrow \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)\}$, whose DPs are $\mathcal{R}_{\mathtt{min}}$ following Example 21, we need $\mathtt{quot}^\sharp\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \succ \mathtt{quot}^\sharp\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y)$ and $\mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) \succeq \mathtt{min}\ x\ y$ and $\mathtt{min}\ x\ \mathtt{0} \succeq x$, as well as $\mathtt{pair}_\iota \succeq \iota$ for all $\iota$. To achieve this, we use the same interpretation as in Example 14, and let $\mathcal{J}_{\mathtt{pair}_\iota} = \max(x, y)$ for all $\iota$.

We have now nearly completed our running example, with only one singular set remaining. To address this last dependency pair, we observe that the use of the function symbol in the $\mathtt{sma}$ rules is innocuous: the size of $\mathtt{sma}\ b\ F\ x$ is bounded by the size of $x$ no matter what kinds of calls the evaluation of $F$ may bring up. It would be nice to ignore the dependency pairs imposed by this relatively harmless function application. To do this, we build on first-order methods once more, and combine usable rules with an *argument filtering*.

▶ **Definition 24** (Argument filtering)**.** *Let a function $\nu$ be given which maps each (marked or unmarked) function symbol $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$ to a subset of $\{1, \ldots, m\}$. If $\nu(\mathtt{f}) = \{i_1, \ldots, i_k\}$ with $i_1 < \cdots < i_k$, then let $\psi_\nu(\mathtt{f}\ s_1 \cdots s_m)$ denote $\mathtt{f}'\ s_{i_1} \cdots s_{i_k}$, where $\mathtt{f}' :: \sigma_{i_1} \Rightarrow \ldots \Rightarrow \sigma_{i_k} \Rightarrow \iota$ is a new function symbol. We define:*

$$\begin{array}{rcl}\overline{\nu}(\mathtt{f}\ t_1 \cdots t_n) & = & \lambda x_{n+1} \ldots x_m.\psi_\nu(\mathtt{f}\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n)\ x_{n+1} \cdots x_m) \text{ if } \mathtt{f} \text{ takes } m \text{ args} \\ \overline{\nu}(x\ t_1 \cdots t_n) & = & x\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n) \\ \overline{\nu}((\lambda x.u)\ t_1 \cdots t_n) & = & (\lambda x.\overline{\nu}(u))\ \overline{\nu}(t_1) \cdots \overline{\nu}(t_n)\end{array}$$

*For a set of rules $\mathcal{R}$, let $\overline{\nu}(\mathcal{R}) = \{\overline{\nu}(\ell) \to \overline{\nu}(r) \mid \ell \to r \in \mathcal{R}\}$, and similar for a set of DPs.*

Essentially, we make sure that all function symbols are maximally applied (by replacing a partially applied function $\mathtt{f}\ s_1 \cdots s_n$ by $\lambda x_{n+1} \ldots x_m.\mathtt{f}\ s_1 \cdots s_n\ x_{n+1} \cdots x_m$), and then remove the arguments that we do not want to consider from their function symbols.

▶ **Lemma 25.** *Suppose $\mathcal{R}$ is finitely branching. Then a set $\mathcal{P}$ is chain-free if $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ where $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succ, \succeq)$ such that: (a) $\ell \succ r$ for all $\ell \Rrightarrow r \in \overline{\nu}(\mathcal{P}_1)$, (b) $\ell \succeq r$ for all $\ell \Rrightarrow r \in \overline{\nu}(\mathcal{P}_2)$ and (c) $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathcal{R})) \cup \mathcal{C}_\epsilon$.*

With this method, we can finally complete our running example.

▶ **Example 26.** We let $\overline{\nu}(\mathtt{sma}^\sharp) = \{2,3\}$ and $\overline{\nu}(\mathtt{f}) = \{1,\dots,m\}$ for all other symbols $\mathtt{f} :: \sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \iota$. Then $\overline{\nu}(\{(\text{R})\}) = \{\mathtt{sma}^\sharp\ F\ (\mathtt{s}\ x) \Rrightarrow \mathtt{sma}^\sharp\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))\}$. Hence, $\mathsf{UR}(\overline{\nu}(\{(\text{R})\}), \overline{\nu}(\mathcal{R})) = \mathsf{UR}(\overline{\nu}(\{(\text{R})\}), \mathcal{R}) = \mathcal{R}_{\mathtt{quot}} \cup \mathcal{R}_{\mathtt{min}}$.

We use the same interpretation for `quot` and `min` as in Example 14, and let $[\![\mathtt{sma}^\sharp\ F\ x]\!] = [\![x]\!]$. Then $[\![\ell]\!] \geq [\![r]\!]$ is satisfied for the usable rules as before, and $[\![\mathtt{sma}^\sharp\ F\ (\mathtt{s}\ x)]\!] = [\![x]\!] + 1 > [\![x]\!] = [\![\mathtt{sma}^\sharp\ F\ (\mathtt{quot}\ x\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})))]\!]$ orients the DP. Hence, our last remaining set $\mathcal{P}$ is chain-free, and the original system is terminating.

In the context of step-wise simplifying a termination problem, *formative rules* are also worth mentioning. These are defined much like usable rules, but from the *left* side of rules and DPs rather than the *right*: $\mathtt{Form}(T, \mathcal{R})$ contains those $\ell \to r \in \mathcal{R}$ such that:

1. $r = \mathtt{f}\ r_1 \cdots r_m$ and there is a term $s \in T$ with $s \trianglerighteq \mathtt{f}\ s_1 \cdots s_m$ for some $s_1, \dots, s_m$, or
2. $r = x\ r_1 \cdots r_m$ and there is a term $s \in T$ with $s \trianglerighteq t$ for some $t$ whose type is the same as the type of $r$, and $t$ is not a free variable in $s$, or
3. there is a term $s \in T$ which is not linear, or has a subterm $\lambda x.t$ with $\mathcal{FV}(t) \cap \mathcal{FV}(s) \neq \emptyset$.

The set $\mathsf{FR}(\mathcal{P}, \mathcal{R})$ of formative rules is the smallest set $O \subseteq \mathcal{R}$ such that $\mathtt{Form}(\mathtt{lhs}(\mathcal{P}), \mathcal{R}) \subseteq O$ and $\mathtt{Form}(\mathtt{lhs}(O), \mathcal{R}) \subseteq O$. Hence, the parallels with usable rules are obvious.

In a more elaborate DP framework, which carries pairs $(\mathcal{P}, \mathcal{R})$ instead of just sets $\mathcal{P}$ and considers more properties for chains than just computability, this definition can be used to remove elements of $\mathcal{R}$ [11, Theorem 58]. In the current, limited DP framework, we can still use formative rules with reduction pairs, for instance by changing requirement (c) in Lemma 25 to: $\ell \succeq r$ for all $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathsf{FR}(\mathcal{P}, \mathcal{R}))) \cup \mathcal{C}_\epsilon$. It seems likely that we can also combine formative rules with an argument filtering, and hence limit interest to $\ell \to r \in \mathsf{UR}(\overline{\nu}(\mathcal{P}), \mathsf{FR}(\overline{\nu}(\mathcal{P}), \overline{\nu}(\mathcal{R}))) \cup \mathcal{C}_\epsilon$. However, this proof currently only exists as a sketch.

Unfortunately, although we can use this method to eliminate some rules, these rules are usually simple; for example, we may throw out the base case of a rule $\mathtt{times}\ \mathtt{0}\ y \to \mathtt{0}$ but not the more complex induction case $\mathtt{times}\ (\mathtt{s}\ x)\ y \to \mathtt{add}\ (\mathtt{s}\ x)\ (\mathtt{times}\ x\ y)$. The primary use case is when the set of sorts can be split, say $\mathbb{S} = A \cup B$, so that the rules of type $A$ do not use any symbols over type $B$; in this case, we may be able to remove all rules of type $B$. However, this does not happen often in practice. Hence, this is not really a core technique.

**Discussion.** The techniques in this section are all direct adaptations of methods for first-order term rewriting, and they are used in a similar way as their first-order counterpart. Yet, there is a clear place for higher-order reasoning, too. Type analysis play a role in both the AFP restriction and the alternative subterm criterion. In the splitting lemma, higher-order reachability analysis can be used to assess whether any reducts of $r\gamma$ are in some $A_i$. The choice of a reduction pair needs to take functional variables and $\beta$-reduction into account.

A critical difference between first-order and higher-order analysis lies in usable rules: case 2 in Definition 20 is not present in the first-order definition, since there variables cannot be applied. But in higher-order rewriting, if any element of $\mathcal{P}$, *or any of its usable rules*, has a subterm $x\ s_0 \cdots s_n$, then all rules are usable. Since a variable of higher type is typically applied eventually (otherwise, why carry it around?), this essentially means that if any rule with a higher-order variable is usable, then all rules are, and Lemma 22 is no improvement over Lemma 13. Effectively: we can only use usable rules in an essentially first-order problem!

Hence, instead of usable rules, Example 23 could have been done using [9], which shows that if the "first-order" part of a higher-order system combined with $\mathcal{C}_\epsilon$ is terminating, then the corresponding DPs may be dropped from $\mathtt{SDP}(\mathcal{R})$. We recover this result with Lemmas 15 and 22: define $\mathsf{FO}$ as the largest subset of $\mathcal{R}$ such that (a) the rules in $\mathsf{FO}$ do not use abstractions, variables of higher type or partially applied function symbols, and (b)

$\mathtt{Use}(\mathtt{rhs}(\mathsf{FO}), \mathcal{R}) \subseteq \mathsf{FO}$. Let $A_2 = \{\mathtt{f}^\sharp\ s_1 \cdots s_n \mid \mathtt{f}$ is the head symbol of the left-hand side of a rule in $\mathsf{FO}\}$, and let $A_1 = \{\mathtt{f}^\sharp\ s_1 \cdots s_m \mid \mathtt{f}$ is a different defined symbol$\}$; by Lemma 15, termination follows if $\mathtt{SDP}(\mathcal{R} \setminus \mathsf{FO})$ and $\mathtt{SDP}(\mathsf{FO})$ are both chain-free. As the usable rules of $\mathtt{SDP}(\mathsf{FO})$ are in $\mathsf{FO}$, we can apply Lemma 22 with $\succ$ the (terminating!) relation $(\rightarrow_{\mathsf{FO} \cup \mathcal{C}_\epsilon} \cup \rhd)^+$ on terms with $\sharp$ marks removed. Hence, it suffices to prove chain-freeness of $\mathtt{SDP}(\mathcal{R} \setminus \mathsf{FO})$.

A similar result appears in [13], but instead of just first-order rules, this paper considers a set $A \subseteq \mathcal{R}$ where both the left- and right-hand sides of rules are patterns. This obviously captures first-order rules, but – due to the more permissive formalism of rewriting used in [13] – also some forms of higher-order rules with particular applications (algebraic effect handlers). To handle $\mathcal{R} \setminus A$, the author of [13] does not use dependency pairs but rather a version of the general schema [4]. There are many similarities between this technique and dependency pairs with the splitting lemma and extended subterm criterion, but the restrictions to apply the general schema do *not* need to apply to $A$. A parallel result in our setting would be that the rules of $A$ would not need to be accessible function passing, yet termination still holds if $\mathtt{SDP}(\mathcal{R} \setminus A)$ is chain-free. It might be worth investigating if this is the case.

These positive results aside, without an argument filtering, usable rules does not give us much else due to the requirement that any variable application makes all rules usable. Unfortunately, this requirement is hard to avoid. Consider for instance the rules $\mathcal{R}_{\mathtt{comp2}}$:

$$\begin{array}{rcl} \mathtt{comp2}\ 0\ (\mathtt{s}\ y) & \rightarrow & \bot \\ \mathtt{comp2}\ (\mathtt{s}\ 0)\ (\mathtt{s}\ y) & \rightarrow & \bot \\ \mathtt{f}\ F\ x\ \bot & \rightarrow & \mathtt{end}\ x \end{array} \qquad \begin{array}{rcl} \mathtt{comp2}\ x\ 0 & \rightarrow & \top \\ \mathtt{comp2}\ (\mathtt{s}\ (\mathtt{s}\ x))\ (\mathtt{s}\ y) & \rightarrow & \mathtt{comp2}\ x\ y \\ \mathtt{f}\ F\ x\ \top & \rightarrow & \mathtt{f}\ F\ (\mathtt{s}\ x)\ (\mathtt{comp2}\ (F\ x)\ x) \end{array}$$

Now, $\rightarrow_{\mathcal{R}_{\mathtt{comp2}} \cup \mathcal{C}_\epsilon}$ is terminating, since $\mathtt{comp2}\ n\ m$ determines whether $n \geq 2 * m$, and the only closed functions from $\mathtt{nat}$ to $\mathtt{nat}$ are built using $\lambda$, $0$, $\mathtt{s}$ and $\mathtt{pair}_{\mathtt{nat}}$. Hence, in the worst case $F$ is linear in its argument, so for large enough $x$, $\mathtt{comp2}\ (F\ x)\ x$ will return $\bot$. However, combining these rules with $\mathtt{double}\ 0 \rightarrow 0$, $\mathtt{double}\ (\mathtt{s}\ x) \rightarrow \mathtt{s}\ (\mathtt{s}\ (\mathtt{double}\ x))$ clearly yields a non-terminating system. Here it is essential that the $\mathtt{double}$ rules are considered usable.

All this means that, if we succeed in applying usable rules – with or without an argument filtering – the corresponding ordering requirements will be essentially first-order (perhaps with some abstractions or unused higher-order variables). When these methods do not apply, there is no obvious way to circumvent the need to orient all rules at once. The same happens when we use *dynamic* instead of *static* DPs, where collapsing pairs often cause the subterm criterion, splitting lemma and usable rules to fail; the static approach is incomplete, so we may need the dynamic approach even on some AFP systems. In the next section we will see how we can also use a modular kind of reasoning to build a suitable reduction pair.

## 5 Incrementally building weakly monotonic interpretations

Although higher-order variations of the *recursive path ordering* [14, 5] have been very succesful in orienting higher-order rules, the current paper instead focuses on *interpretations*. The reason for this is twofold. First, the static dependency pair approach already captures many of the same advantages as higher-order RPO, since both methods are based on the same proof technique (computability). The second, and main, reason is that, unlike RPO, an interpretation-based ordering for a large set of rules can usually be built step by step.

Weakly monotonic interpretations do not provide a complete proof method: there are terminating systems that cannot be ordered with interpretations. Nevertheless, it has the potential to be very powerful – if we choose the sets $\mathcal{A}_\iota$ right. In the examples so far, we

have let $\mathcal{A}_\iota = \mathbb{N}$ for all sorts, but this is fundamentally limiting. For example, if other rules impose that $[\![\mathtt{s}\ x]\!] > [\![x]\!]$, we cannot orient $\mathtt{inc}\ \mathtt{0} \to \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ \mathtt{0}))$. Instead, following an approach for complexity in [17], we will map terms to *tuples* of numbers.

Intuitively, we assign to all sorts a variety of numbers to indicate different measures of *size*. For example, a string of `a`s and `b`s might be mapped to the number of `a`s, the number of `b`s, and the total length. Then we express for each rule how it affects the size measures. This is a semantic technique: rather than only looking at the shape of rules, the best results are typically obtained by modelling our interpretation to the intended meaning of the rules.

We left Section 4 with some techniques that *often*, but not *always* allow us to cut a termination proof into bite-sized chunks. In the remaning cases, we must orient a large number of rules and – typically – a small number of DPs using a reduction pair. To find an interpretation (following Definition 11) that lets us do so, we will use the following procedure:

1. We choose an initial set $\mathcal{A}_\iota$ for each sort, along with an intuitive meaning, and define $\mathcal{J}_{\mathtt{f}}$ for all constructor symbols `f` according to this meaning.
2. We divide the defined symbols into sets $\mathcal{D}_1, \dots, \mathcal{D}_n$ such that for each $\mathtt{f} \in \mathcal{D}_i$, all the function symbols occurring in the rules defining `f` are either constructors or in $\mathcal{D}_1 \cup \cdots \cup \mathcal{D}_i$.
3. For all $i$ (starting with 1 going up to $n$), we find interpretations for the symbols in $\mathcal{D}_i$ so that $[\![\ell]\!] \sqsupseteq [\![r]\!]$; we strive to make them *as tight as possible*, to make later rules easier.
4. If we find that some rule of sort $\iota$ cannot be oriented, we extend $\mathcal{A}_\iota$ with an additional measure that does make this possible (if we can). We return to the previous step, updating the interpretations we already had to take the new measure into account.
5. When all rules are oriented, we find interpretations for the DPs in the same way.

This approach has not been formalised or implemented; rather, the goal is to present *ideas*; to hopefully lay the foundation for an automated approach in the future.

Let us explore how the procedure works by applying it to a large example.

**Preparation.** Let $\mathcal{R}$ consist of the rules in Example 2 combined with the following:

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{hd}\ (\mathtt{cons}\ x\ a) & \to & x & \mathtt{len}\ \mathtt{nil} & \to & \mathtt{0} \\
\mathtt{id}\ x & \to & x & \mathtt{len}\ (\mathtt{cons}\ x\ a) & \to & \mathtt{s}\ (\mathtt{len}\ a) \\
\mathtt{twice}\ F\ x & \to & F\ (F\ x) & \mathtt{H}\ (\mathtt{s}\ x) & \to & \mathtt{H}\ (\mathtt{twice}\ \mathtt{id}\ x)
\end{array}$$

For $\mathcal{P} = \{\mathtt{H}^\sharp\ (\mathtt{s}\ x) \Rrightarrow \mathtt{H}^\sharp\ (\mathtt{twice}\ \mathtt{id}\ x)\} \subseteq \mathtt{SDP}(\mathcal{R})$, all rules are usable, the subterm criterion cannot be applied, and there is no argument filtering that stops all rules from being usable and yet allows us to strictly orient the single dependency pair. Hence, as we noted before, we need to find an interpretation to show $[\![\ell]\!] \succeq [\![r]\!]$ for a large number of rules (all rules in the system), and $[\![\ell]\!] \succ [\![r]\!]$ for a small number of DPs (the single element of $\mathcal{P}$).

So let us begin! Following step 1, we assign an intuitive measure to each type: terms of type `nat` are mapped to the corresponding number, lists to their largest element, and booleans to 0 or 1: $\mathcal{A}_{\mathsf{nat}} = \mathcal{A}_{\mathsf{list}} = (\mathbb{N}, >, \geq)$, $\mathcal{A}_{\mathsf{bool}} = (\{0,1\}, >, \geq)$. This corresponds with:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathcal{J}_{\mathtt{0}} & = & 0 & \mathcal{J}_{\mathtt{nil}} & = & 0 & \mathcal{J}_{\bot} & = & 0 \\
\mathcal{J}_{\mathtt{s}}(x) & = & x+1 & \mathcal{J}_{\mathtt{cons}}(x,a) & = & \max(x,a) & \mathcal{J}_{\top} & = & 1
\end{array}$$

We will handle the defined symbols in the following order: {`id`}, {`twice`}, {`min`}, {`quot`}, {`sma`}, {`hd`}, {`ack`}, {`map`}, {`mkbig`}, {`mkdiv`}, {`len`}, {`fold`}, {`inc`}, {`double`, `exp`}. This satisfies the requirement on the order of symbols, and is otherwise arbitrary.

**The straightforward part.** Following step 3, we will repeatedly interpret one or more defined symbols whose rules only depend on each other and symbols that already have an interpretation. To start, if $\mathcal{J}_{\mathtt{id}}(x) = x$ clearly $[\![\mathtt{id}\ x]\!] = [\![x]\!]$. The rule defining `id` is oriented, and since we have an equality, this interpretation is as tight as possible. We can achieve the same for `twice`: with $\mathcal{J}_{\mathtt{twice}}(F, x) = F(F(x))$ we have $[\![\ell]\!] = [\![r]\!]$ for the corresponding rule.

Unfortunately, we cannot achieve equality for `min`. Due to the monotonicity requirement, we cannot have $\mathcal{J}_{\mathtt{min}}(x, y) = x - y$, which would give a tight interpretation. For the current choice of $(\mathcal{A}_{\mathsf{nat}}, \sqsupset_{\mathsf{nat}}, \sqsupseteq_{\mathsf{nat}})$, the best we can do is $\mathcal{J}_{\mathtt{min}}(x, y) = x$. With this choice, $[\![\mathtt{min}\ x\ \mathtt{0}]\!] = [\![x]\!]$, and $[\![\mathtt{min}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)]\!] = [\![x]\!] + 1 > [\![x]\!] = [\![\mathtt{min}\ x\ y]\!]$, so the rules are oriented.

Next is `quot`. Since we already know $\mathcal{J}_{\mathtt{f}}$ for all other symbols in the two `quot` rules, the requirements are: $[\![\mathtt{quot}\ \mathtt{0}\ (\mathtt{s}\ y)]\!] = \mathcal{J}_{\mathtt{quot}}(0, y+1) \geq 0 = \mathcal{J}_{\mathtt{0}} = [\![\mathtt{0}]\!]$, and $[\![\mathtt{quot}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y)]\!] = \mathcal{J}_{\mathtt{quot}}(x+1, y+1) \geq \mathcal{J}_{\mathtt{quot}}(x, y+1) + 1 = [\![\mathtt{s}\ (\mathtt{quot}\ (\mathtt{min}\ x\ y)\ (\mathtt{s}\ y))]\!]$. This is easily satisfied with $\mathcal{J}_{\mathtt{quot}}(x, y) = x$ (which is tight, as the left- and right-hand side are equal in both rules).

Similarly, the requirements for `sma` are: $\mathcal{J}_{\mathtt{sma}}(b, F, 0) \geq 0$ and $\mathcal{J}_{\mathtt{sma}}(1, F, x+1) \geq x + 1$ and $\mathcal{J}_{\mathtt{sma}}(0, F, x+1) \geq \mathcal{J}_{\mathtt{sma}}(F(x), F, x)$. The simplest solution is $\mathcal{J}_{\mathtt{sma}}(b, F, x) = x$. To orient $\mathtt{hd}\ (\mathtt{cons}\ x\ a) \to x$, we let $\mathcal{J}_{\mathtt{hd}}(x) = x$; this suffices because $\max(x, a) \geq x$, and is optimal.

**Beyond polynomials.** When adressing `ack`, we run into some trouble: thus far, all our interpretation functions $\mathcal{J}_{\mathtt{f}}$ have been bounded by polynomials, but these rules implement the Ackermann function which grows much faster than any polynomial. However, there is no need to limit interest to polynomials. Indeed, the three rules provide a recursive specification:

$$\begin{array}{lcl} \mathtt{ack}\ \mathtt{0}\ y & = & \mathtt{s}\ y \end{array} \qquad \begin{array}{rcl} \mathtt{ack}\ (\mathtt{s}\ x)\ \mathtt{0} & = & \mathtt{ack}\ x\ (\mathtt{s}\ \mathtt{0}) \\ \mathtt{ack}\ (\mathtt{s}\ x)\ (\mathtt{s}\ y) & = & \mathtt{ack}\ x\ (\mathtt{ack}\ (\mathtt{s}\ x)\ y) \end{array}$$

We can see by the recursive path ordering that this is terminating, and since it is a non-overlapping constructor system, it is confluent. Hence, we can define *Ack* as a function from $\mathbb{N}$ to $\mathbb{N}$, and choose $\mathcal{J}_{\mathtt{ack}}(x, y) = \mathit{Ack}(x, y)$. Then obviously all three `ack` rules are oriented.

We orient `map` by $\mathcal{J}_{\mathtt{map}}(F, a) = F(a)$: by weak monotonicity of $F$ we have $F(\max(x, a)) \geq F(x)$. Intuitively, applying $F$ to *some* element of the list cannot be greater than $F$(largest element). To orient the `mkbig` rules, we must have $\mathcal{J}_{\mathtt{mkbig}}(a, x) \geq \mathcal{J}_{\mathtt{map}}(\mathcal{J}_{\mathtt{ack}}(x), a) = \mathit{Ack}(x, a)$, so we choose $\mathtt{mkbig}(a, x) = \mathit{Ack}(x, a)$. For `mkdiv`, we let $\mathcal{J}_{\mathtt{mkdiv}}(x, a) = \mathcal{J}_{\mathtt{quot}}(a, x) = a$.

**Backtracking.** We are in trouble again when trying to orient the `len` rule: the interpretation of the constructors imposes $\mathcal{J}_{\mathtt{len}}(0) = 0$ and $\mathcal{J}_{\mathtt{len}}(\max(x, a)) \geq 1 + \mathcal{J}_{\mathtt{len}}(a)$. The latter is not satisfiable since (for $x = a$) it implies $\mathcal{J}_{\mathtt{len}}(a) \geq 1 + \mathcal{J}_{\mathtt{len}}(a)$. The problem lies in the choice for $\mathcal{J}_{\mathtt{cons}}$, which does not give enough information. Similarly, if we had chosen $\mathcal{J}_{\mathtt{cons}}(x, a) = a + 1$ (so mapping a list to its length), we could have oriented the `len` rules but not `hd`.

Hence, we are at Step 4: extending the sort interpretations. We can keep $\mathcal{A}_{\mathsf{nat}}$ unchanged, but let us take $\mathcal{A}_{\mathsf{list}} := \mathbb{N}^2$, mapping a list of numbers to the pair of its greatest argument and its length (ordered with $\geq^{\times}$ as described in Section 2). The constructors are mapped to:

$$\mathcal{J}_{\mathtt{nil}} = \langle 0, 0 \rangle \qquad \mathcal{J}_{\mathtt{cons}}(x, \langle m, l \rangle) = \langle \max(x, m), l + 1 \rangle$$

This follows the intended meaning of the sort. In line with Step 4 we now need to go back and update all interpretations for the new target set $\mathcal{A}_{\mathsf{nat}}$ and the new interpretations for `nil` and `cons`. However, this turns out to be quite easy. Note that in the interpretations of the constructors, the original choices 0 and $\max(x, a))$ are still present, in the first component.

Similarly, the interpretations for the defined symbols are adapted by (a) replacing any list variable by its first component, and (b) adding a length component to the interpretation for the defined symbols of a type $\vec{\sigma} \Rightarrow \mathsf{list}$, so that $[\![\ell]\!]_2 \geq [\![r]\!]_2$ for the relevant rules. This yields:

| **Original:** | **Update:** |
|---|---|
| $\mathcal{J}_{\mathtt{hd}}(a) = a$ | $\mathcal{J}_{\mathtt{hd}}(\langle m, l\rangle) = m$ |
| $\mathcal{J}_{\mathtt{map}}(F, a) = F(a)$ | $\mathcal{J}_{\mathtt{map}}(F, \langle m, l\rangle) = \langle F(m), l\rangle$ |
| $\mathcal{J}_{\mathtt{mkbig}}(a, x) = Ack(x, a)$ | $\mathcal{J}_{\mathtt{mkbig}}(\langle m, l\rangle, x) = \langle Ack(x, m), l\rangle$ |
| $\mathcal{J}_{\mathtt{mkdiv}}(a, x) = a$ | $\mathcal{J}_{\mathtt{mkdiv}}(\langle m, l\rangle, x) = \langle m, l\rangle$ |

The interpretations for `id`, `twice`, `min`, `quot`, `sma` and `ack` are unchanged as $\mathsf{list}$ does not occur in their type. We can orient the `len` rules using $\mathcal{J}_{\mathtt{len}}(\langle m, l\rangle) = l$.

Continuing our example, we orient $\mathcal{R}_{\mathtt{fold}}$ with $\mathcal{J}_{\mathtt{fold}}(F, x, \langle m, l\rangle) = (d \mapsto F(d, m))^l(x)$, so using repeated function application. To see that this works, denote $[\![a]\!] = \langle m, l\rangle$. Then:

$$\begin{array}{rcl} [\![\mathtt{fold}\ F\ x\ (\mathtt{cons}\ y\ a)]\!] & = & (d \mapsto F(d, \max(y, m)))^{l+1}(x) \\ & = & (d \mapsto F(d, \max(y, m)))^{l}((d \mapsto F(d, \max(y, m)))(x)) \\ & = & (d \mapsto F(d, \max(y, m)))^{l}(F(x, \max(y, m))) \\ & \geq & (d \mapsto F(d, m))^{l}(F(x, y)) \text{ by weak monotonicity of } F \\ & = & [\![\mathtt{fold}\ F\ (F\ x\ y)\ a]\!] \end{array}$$

**Non-numeric interpretations.** As observed before, we cannot orient the `inc` rule if $[\![\mathtt{s}\ x]\!] > [\![x]\!]$, which is currently the case. To handle this problem, we must backtrack again, and update $\mathcal{A}_{\mathsf{nat}}$. Let $X = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ with $\mathtt{a} > \mathtt{b}$ and $\mathtt{a} > \mathtt{c}$. We let $\mathcal{A}_{\mathsf{nat}} = \mathbb{N} \times X$, and set:

$$\begin{array}{rclrcl} \mathcal{J}_{\mathtt{0}} & = & \langle 0, \mathtt{b}\rangle & \mathcal{J}_{\mathtt{s}}(\langle n, e\rangle) & = & \langle n+1, \mathtt{c}\rangle \\ \mathcal{J}_{\mathtt{nil}} & = & \langle 0, 0\rangle & \mathcal{J}_{\mathtt{cons}}(\langle n, e\rangle, \langle m, l\rangle) & = & \langle \max(n, m), l+1\rangle \end{array}$$

(Note that we had to adapt $\mathcal{J}_{\mathtt{cons}}$ because it takes a $\mathsf{nat}$ as argument, but the interpretation is essentially unchanged: the new component is simply discarded.)

With this interpretation, $[\![\mathtt{s}\ \mathtt{0}]\!] = \langle 1, \mathtt{c}\rangle \not\sqsupset_{\mathsf{nat}} \langle 0, \mathtt{b}\rangle = [\![\mathtt{0}]\!]$. Now we can orient the `inc` rule using: $\mathcal{J}_{\mathtt{inc}}(x, e) =$ "if $e = \mathtt{c}$ then 0 else 1". Then $[\![\mathtt{inc}\ \mathtt{0}]\!] = 1 = \mathtt{s}\ (\mathtt{inc}\ (\mathtt{s}\ \mathtt{0}))$. We update the existing interpretations by replacing references to a natural number $x$ by its first component, and letting the second component of every defined symbol be $\mathtt{a}$:

$$\begin{array}{rclrcl} \mathcal{J}_{\mathtt{id}}(\langle n, e) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{twice}}(F, \langle n, e\rangle) & = & F\ (F\ \langle n, e\rangle) \\ \mathcal{J}_{\mathtt{min}}(\langle n, e\rangle, \langle m, i\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{ack}}(\langle n, e\rangle) & = & \langle Ack(n), \mathtt{a}\rangle \\ \mathcal{J}_{\mathtt{quot}}(\langle n, e\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{map}}(F, \langle m, l\rangle) & = & \langle F(\langle m, \mathtt{a}\rangle), l\rangle \\ \mathcal{J}_{\mathtt{sma}}(b, F, \langle n, e\rangle) & = & \langle n, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{mkbig}}(\langle m, l\rangle, \langle n, e\rangle) & = & \langle Ack(n, m), l\rangle \\ \mathcal{J}_{\mathtt{hd}}(\langle m, l\rangle) & = & \langle m, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{mkdiv}}(\langle m, l\rangle, \langle n, e\rangle) & = & \langle m, l\rangle \\ \mathcal{J}_{\mathtt{len}}(\langle m, l\rangle) & = & \langle l, \mathtt{a}\rangle & \mathcal{J}_{\mathtt{fold}}(F, \langle n, e\rangle, \langle m, l\rangle) & = & (d \mapsto F(d, \langle m, \mathtt{a}\rangle))^l(\langle n, e\rangle) \end{array}$$

**Mutually recursive symbols.** To handle the mutually recursive symbols `double` and `exp`, we can either find assignments for $\mathcal{J}_{\mathtt{exp}}$ and $\mathcal{J}_{\mathtt{double}}$ at the same time, or use a trick: the system is essentially unchanged if we replace these rules by the following:

$$\begin{array}{rclrcl} \mathtt{exp}\ \mathtt{0}\ y & \to & y & \mathtt{exp}\ (\mathtt{s}\ x)\ y & \to & \mathtt{double}\ x\ y\ \mathtt{0}\ \mathtt{exp} \\ \mathtt{double}\ x\ \mathtt{0}\ z\ F & \to & F\ x\ z & \mathtt{double}\ x\ (\mathtt{s}\ y)\ z\ F & \to & \mathtt{double}\ x\ y\ (\mathtt{s}\ (\mathtt{s}\ z))\ F \end{array}$$

Now `double` and `exp` are no longer mutually recursive, and can be handled separately. For `double`, we can choose $\mathcal{J}_{\mathtt{double}}(x, \langle y, u\rangle, \langle z, e\rangle, F) := F(x, \langle z + 2 * y, \mathtt{a}\rangle)$. Using this, the requirements for `exp` evaluate to $\mathcal{J}_{\mathtt{exp}}(\langle 0, \mathtt{b}\rangle, y) \sqsupseteq_{\mathsf{nat}} y$ and $\mathcal{J}_{\mathtt{exp}}(\langle x+1, \mathtt{c}\rangle, \langle y, e\rangle) \sqsupseteq_{\mathsf{nat}}$

$\mathcal{J}_{\texttt{exp}}(\langle x,u\rangle,\langle 2*y,\texttt{a}\rangle)$. This is satisfied with $\mathcal{J}_{\texttt{exp}}(\langle x,u\rangle,\langle y,e\rangle) = \langle 2^x * y,\texttt{a}\rangle$. Now we can find an interpretation for the *original* definition of `double` by replacing $F$ by $\mathcal{J}_{\texttt{exp}}$; this gives $\mathcal{J}_{\texttt{double}}(\langle x,i\rangle,\langle y,u\rangle,\langle z,e\rangle) = \langle 2^x * (z+2*y),\texttt{a}\rangle$.

In this case, we only had two mutually recursive symbols, so the separation was perhaps unnecessary. However, to handle a large group of mutually recursive rules, this idea may be indispensible to split it into manageable chunks. Note also that we used the higher-order capabilities of interpretations, even though the `exp` and `double` rules are first-order.

**Finishing up.** The last rule, $\texttt{H}\,(\texttt{s}\;x) \to \texttt{H}\,(\texttt{twice}\;\texttt{id}\;x)$, can be handled by choosing $\mathcal{J}_{\texttt{H}}(x) = 0$. Now, having $[\![\ell]\!] \sqsupseteq [\![r]\!]$ for all rules, we move on to step 5 of the procedure. We let $\mathcal{A}_{\mathsf{dp}} = \mathbb{N}$ and orient the DP by choosing $\mathcal{J}_{\texttt{H}^\sharp}(\langle x,e\rangle = x$. Then, using $p_1$ to denote the first element of a pair $p$, we have $[\![\texttt{H}\;(\texttt{s}\;x)]\!] = [\![x]\!]_1 + 1 > [\![x]\!]_1 = \mathcal{J}_{\texttt{id}}(\mathcal{J}_{\texttt{id}}(x))_1 = [\![\texttt{H}\;(\texttt{twice}\;\texttt{id}\;x)]\!]$ as required. Hence, the termination proof of the extended system is complete.

It is worth noting that there are many similarities between dependency pairs and this incremental procedure for interpretations. Dividing the function symbols in groups based on mutual dependencies also happens in the splitting lemma, and handling them in order so that the dependencies for a rule $\texttt{f}\;\vec{\ell} \to r$ have been computed before $\mathcal{J}_{\texttt{f}}$ is reminiscent of usable rules. Non-numeric interpretations like $\{\texttt{a},\texttt{b},\texttt{c}\}$ can take the same role as reachability analysis in the splitting lemma. Also, *strongly* monotonic tuple interpretations (used without dependency pairs) avoid the problem that $\texttt{f}\;\vec{x} \succeq x_i$ of Example 5, and can handle $\mathcal{R}_{\texttt{quot}} \cup \mathcal{R}_{\texttt{min}}$.[17]. Hence, tuple interpretations transpose DP-like reasoning to the level of rules rather than dependency pairs. In future work it might be possible to define a similar reasoning approach as the DP framework, but based on interpretations rather than dependency pairs. This may offer a powerful tool for complexity analysis similar to the DP framework for termination.

## Formalisation and implementation

The procedure above illustrates how a human can find tuple interpretations in a systematic way. However, to be practically usable for systems with thousands of rules, the approach needs to be automated – and to achieve that, there is a lot of work still to be done.

- The methods to find individual interpretations should be automated. This could be done using an encoding to SAT or SMT [7, 8, 10, 24], but the existing techniques will have to be extended to for instance support repeated function application $F^n(x)$.
- The use of interpretations to sets like $\{\texttt{a},\texttt{b},\texttt{c}\}$, which we used as a kind of reachability check, should be formalised and explored more deeply. The same holds for defining functions like *Ack* based on a given terminating and confluent subset of $\mathcal{R}$.
- The process to adapt existing interpretations when $\mathcal{A}_\iota$ is expanded should be formalised. To be precise, we would like to find a systematic way to modify an interpretation function $\mathcal{J}$ so that previously proven inequalities $[\![\ell]\!] \sqsupseteq [\![r]\!]$ are preserved either directly if $\ell :: \kappa \neq \iota$, or in the first component (i.e., $[\![\ell]\!]_1 \sqsupseteq_\iota [\![r]\!]_1$) if $\ell :: \iota$. This was straightforward in all examples that we have seen, but it is not easy to define an algorithm. We *conjecture* that this can be done in general, but it may require also changing $\mathcal{A}_\kappa$ for some other sorts.
  If the conjecture is false, we could alternatively do a true backtracking step, and recompute all interpretations. Doing this means repeatedly discarding prior work, but it has the advantage that, with the new information, we may be able to find tighter interpretations. (For example, with $[\![\mathsf{nat}]\!] = \mathbb{N} \times \{\texttt{a},\texttt{b},\texttt{c}\}$, there is a smaller choice for $\mathcal{J}_{\texttt{min}}$.)

- When splitting a group of mutually recursive symbols, the choice of *which* function symbol to give an extra argument to matters. In the example, replacing the exp rules by exp 0 $y\ F \to y$ and exp (s $x$) $y\ F \to F\ x\ y$ 0 would not have given the same good result, since there is no perfectly tight interpretation for these rules. Hence, we should either find a good heuristic to choose the symbol, or use a procedure based on trial and error.

## 6 Conclusions

In this paper, we explored a group of methods that can be combined to build termination proofs for many large higher-order TRSs, in an incremental way. The foundation is the *static DP approach*, with techniques lifted from the first-order setting but adapted to higher-order rewriting: the splitting lemma, two subterm criteria and two usable rules lemmas. As a reduction pair, we considered weakly monotonic interpretations to *tuples*, an idea originating in complexity analysis which avoids many limitations of interpretations to $\mathbb{N}$. Most of the theory is not new (though it is adapted to a different formalism), but is used in a new way, to hopefully provide insights on the challenge of large higher-order termination problems.

A part of the techniques discussed in this paper have been implemented in Wanda [15], but not yet usable rules with respect to an argument filtering, or any form of tuple interpretations. An obvious goal for future work is to complete this implementation, and to formalise and implement the ideas of Section 5. In addition, an important goal is to transpose the methodology (and implementation) to functional programming languages. This would also allow us to investigate the power of the framework on real systems. While the termination problem database [22] does contain large systems, these are invariably first-order systems with only a few, mostly very simple, higher-order rules.

Finally, there are many ways to improve the DP framework. This could take the form of lifting more ideas from the first-order setting, recognising more situations where not all rules need to be usable (such as the DP for the H rule), or finding a way to weaken or drop the AFP restriction, for instance by combining static and dynamic dependency pairs.

## References


1. T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In *Proc. RTA '05*, volume 3467 of *LNCS*, pages 120–134, 2005.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3. F. Blanqui. Termination and confluence of higher-order rewrite systems. In *Proc. RTA '00*, volume 1833 of *LNCS*, pages 47–61, 2000.
4. F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.
5. F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proc. CSL '08*, volume 5213 of *LNCS*, pages 1–14, 2008.
6. F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.
7. C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.
8. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT '07*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.

9. C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS '11*, volume 6989 of *LNAI*, pages 147–162, 2011.
10. C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA '12*, volume 15 of *LIPIcs*, pages 176–192, 2012.
11. C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proc. ESOP '19*, volume 11423 of *LNCS*, pages 752–782, 2019.
12. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.
13. M. Hamana. Modular termination for second-order computation rules and application to algebraic effect handlers. Arxiv preprint arXiv:1912.03434, 2019.
14. J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proc. LICS '99*, IEEE, pages 402–411, 1999.
15. C. Kop. WANDA – a higher-order termination tool. In *Proc. FSCD 20*, volume 167 of *LIPIcs*, pages 36:1–36:19, 2020.
16. C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012.
17. C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD '21*, volume 195 of *LIPIcs*, pages 31:1–31:22. Dagstuhl, 2021.
18. K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.
19. K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 18(5):407–431, 2007.
20. J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proc. HOA 94*, volume 816 of *LNCS*, pages 305–325, 1994.
21. Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987.
22. Wiki. Termination Problems DataBase (TPDB). URL: `http://termination-portal.org/wiki/TPDB`.
23. Wiki. The International Termination Competition (TermComp), 2018. URL: `http://termination-portal.org/wiki/Termination_Competition`.
24. A. Yamada. Multi-dimensional interpretations for termination of term rewriting. In *Proc. CADE 21*, volume 12699 of *LNAI*, pages 273–290, 2021.

# A Methodology for Designing Proof Search Calculi for Non-Classical Logics

**Alwen Tiu**
School of Computing, The Australian National University, Canberra, Australia

## Abstract

In this talk I present a methodology for designing proof search calculi for a wide range of non-classical logics, such as modal and tense logics, bi-intuitionistic (linear) logics and grammar logics. Most of these logics cannot be easily formalised in the traditional Gentzen-style sequent calculus; various structural extensions to sequent calculus seem to be required. One of the more expressive extensions of sequent calculus is Belnap's display calculus, which allows one to formalise a very wide range of logics and which provides a generic cut-elimination method for logics formalised in the calculus. The generality of display calculus derives partly from the pervasive use of structural rules to capture properties of the underlying semantics of the logic of interest, such as various frame conditions in normal modal logics, that are not easily captured by introduction rules alone. Unlike traditional sequent calculi, the subformula property in display calculi does not typically give an immediate bound on the search space (assuming contraction is absent) in proof search, as new structures may be created and their creation may not be driven by any introduction rules for logical connectives. This line of work started out as an attempt to "tame" display calculus, to make it more proof search friendly, by eliminating or restricting the use of structural rules. Two key ideas that make this possible are the adoption of *deep inference*, allowing inference rules to be applied inside a nested structure, and the use of *propagation rules* in place of structural rules. A brief survey of the applications of this methodology to a wide range of logics is presented, along with some directions for future work.

**2012 ACM Subject Classification** Theory of computation → Proof theory

**Keywords and phrases** Proof theory, Sequent calculus, Display calculus, Nested sequent calculus, Deep inference

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.2

**Category** Invited Talk

## 1 Summary

Non-classical logics, such as modal logics and intermediate logics, have generally been challenging to formalise in the traditional Gentzen's sequent calculi. This has motivated the development of a variety of structural extensions of sequent calculi as alternative proof-theoretic formalisms for these logics. Notable formalisms include display calculi [14], hypersequent calculi [1], tree-hypersequent calculi [19], nested sequent calculi [15, 2], the calculus of structures [12] and labelled sequent calculi [6, 17]. Among these formalisms, display calculi and labelled sequent calculi are perhaps the more general ones, allowing one to design proof systems for a wide range of non-classical logics that satisfy cut admissibility. In display calculus, this is achieved by essentially defining a *structural connective* for each logical connective, and internalising the underlying semantic conditions (e.g., frame conditions in modal logics) into structural rules manipulating the relevant structural connectives. Similarly, in labelled sequent calculi, the labels in a sequent and their relations can be seen as a representation of Kripke frames in the underlying semantics of the logics, and the "structural rules" manipulating these labels and relations are derived directly from the frame conditions characterising the logics.

© Alwen Tiu;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 2; pp. 2:1–2:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, the generality and the ease in which one represents logics in these calculi come with a price of the loss of some of the more appealing features of Gentzen sequent calculi from the perspective of proof search. The *subformula property* in the traditional sequent calculi provides an immediate way to bound the search space in proof search (assuming contraction is absent), but this is not the case for display calculi or labelled calculi in general. This is a consequence of the use of extended structural rules, which can potentially create structures of an arbitrary size (reading the inference rules bottom up), independently of the (sub)formulas in the end sequent. Another property that arises naturally from a formulation of a logic in the traditional sequent calculus is what I call the *separation property* – given a sequent calculus for a logic, one can extract a sound and complete proof system for any of its sublogics (defined by a selection of connectives) by simply selecting the introduction rules for the connectives defining the sublogic. This property is generally difficult to prove directly in display calculi, as proofs of a formula in a sublogic may require the use of structural connectives that sit outside the sublogic.

In this talk, I present a methodology for designing proof calculi for non-classical logics, for which both the subformula property and the separation property hold. This methodology is based on a refinement process, starting with a "display-like" calculus for a logic, and ending with a nested sequent calculus for the same logic. The syntactic framework for the refinement is that of nested sequent calculus. We generalise the notion of a traditional (one-sided or two-sided) sequent to a tree of sequents, and adopt display-like structural rules that act on the tree of sequents. For example, the familiar *display rules* [14] in our setting becomes essentially a rule that rotates the tree structure of a nested sequent [10].

Our methodology proceeds in three phases. In the first phase, given a logic of interest, we first extend the logic by adding the adjoints of its connectives (if needed). For example, if the logic of interest is an intuitionistic logic, then we will extend it by an exclusion (or subtraction) connective; if it is a modal logic, we extend it to tense logic. We then design a display calculus for the extended logic and produce a *shallow* nested sequent calculus (where introduction rules can be applied only to the root sequent in a nested sequent). A shallow nested calculus is for most part a notational variant of the display calculus. Provided that the shallow nested calculus satisfies Belnap's eight conditions [14], we get cut-elimination for free. The structural rules in the shallow calculus consist of the *internal* structural rules (that change the structures within a sequent, e.g., contraction/weakening) and the *external* structural rules (that change the shape of the tree of a nested sequent, e.g., display postulates and various rules that correspond to frame conditions in modal logic).

In the second phase, we transform the shallow nested sequent calculus into a *deep* nested sequent calculus, where inference rules (including introduction rules) can be applied to any sequent in the nested sequent. A key technical requirement for the deep calculus is that the *only* rules that are allowed change the structure of a nested sequent (i.e., the tree-shape of the nested sequent) are introduction rules. This means in particular that *all* external structural rules are absent in the deep calculus. In place of external structural rules, we introduce *propagation rules* [8, 10] into the deep calculus. These propagation rules determine how formulas in a sequent in the tree of sequents can be propagated to other sequents in the same tree. An important consequence of the absence of external structural rules is that in the proof of a formula, every (logical or structural) connective occurring in the proof also occurs in the formula. This gives us immediately the separation property.

In the last phase, we obtain the deep nested sequent calculus for the logic we started with by simply omitting the introduction rules for the connectives that are not in the logic; by the separation property, this gives us a sound and complete proof system for our logic.

Over the past decade or so, my collaborators and I have applied this methodology to design proof calculi that are amenable to proof search for various non-classical logics. Our early work focused on classical propositional tense logics [8, 10], showing that we can obtain sound and complete cut-free proof systems for all modal logics that can be characterised using *path axioms* [10], which subsume all modal logics in the modal logic cube. This result naturally extends to multi-modal logics, which we have demonstrated by giving cut-free proof systems for a family of grammar logics [21] and their proof search procedures. We applied the same methodology to solve the problem of finding a cut-free proof system for bi-intuitionistic logic [7, 20], which had evaded previous attempts [18]. This was later extended to a version of bi-intuitionistic tense logic [9], which contains an intuitionistic modal logic as its subsystem. Lastly, we have also applied this methodology to design a proof system [4] for full intuitionistic linear logic (FILL) [13] and prove its NP-completeness.

Although our methodology has been successfully applied to design proof calculi for a wide range of logics, it is currently not clear what the limit of its applicability is. We know, for example, modal logics admitting pseudo-transitive axioms of the form $\Box^m p \to \Box^n p$, for $m, n > 1$, do not seem to be expressible in the deep nested sequent calculus without any external structural rules.

There are indications that our deep nested calculi may allow for a more syntax directed proof of the interpolation theorem for a wide range of modal/tense logics and bi-intuitionistic logic [16]. However, proving interpolation for FILL in the deep nested sequent calculus remains a challenge and is a subject of an on-going research.

Our methodology is mainly aimed at bridging display calculi and (deep) nested sequent calculi. It will be interesting to see how this methodology can be generalised to design proof search calculi in a different syntactic framework. For this, we can leverage on existing work on relating different formalisms [5, 11, 3].

## References


1. Arnon Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991. doi:10.1007/BF01531058.
2. Kai Brünnler. Deep sequent systems for modal logic. *Arch. Math. Log.*, 48(6):551–577, 2009.
3. Agata Ciabattoni, Tim S. Lyon, Revantha Ramanayake, and Alwen Tiu. Display to labeled proofs and back again for tense logics. *ACM Trans. Comput. Log.*, 22(3):20:1–20:31, 2021. doi:10.1145/3460492.
4. Ranald Clouston, Jeremy E. Dawson, Rajeev Goré, and Alwen Tiu. Annotation-free sequent calculi for full intuitionistic linear logic. In *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 197–214. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/LIPIcs.CSL.2013.197.
5. Melvin Fitting. Nested sequents and prefixed tableaus. In Martin Giese and Roman Kuznets, editors, *TABLEAUX 2011 - Workshops, Tutorials, and Short Papers, Bern, Switzerland, July 4-8, 2011*, volume IAM-11-002 of *Technical Report*, page 69, 2011.
6. Dov M Gabbay. *Labelled deductive systems*, volume 33 of *Oxford Logic guides*. Clarendon Press/Oxford Science Publications, 1996.
7. Rajeev Goré, Linda Postniece, and Alwen Tiu. Cut-elimination and proof-search for bi-intuitionistic logic using nested sequents. In Carlos Areces and Robert Goldblatt, editors, *Advances in Modal Logic 7, papers from the seventh conference on "Advances in Modal Logic," held in Nancy, France, 9-12 September 2008*, pages 43–66. College Publications, 2008. URL: http://www.aiml.net/volumes/volume7/Gore-Postniece-Tiu.pdf.
8. Rajeev Goré, Linda Postniece, and Alwen Tiu. Taming displayed tense logics using nested sequents with deep inference. In *TABLEAUX*, volume 5607 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2009.

9 Rajeev Goré, Linda Postniece, and Alwen Tiu. Cut-elimination and proof search for bi-intuitionistic tense logic. In Lev D. Beklemishev, Valentin Goranko, and Valentin B. Shehtman, editors, *Advances in Modal Logic 8, papers from the eighth conference on "Advances in Modal Logic," held in Moscow, Russia, 24-27 August 2010*, pages 156–177. College Publications, 2010. URL: `http://www.aiml.net/volumes/volume8/Gore-Postniece-Tiu.pdf`.

10 Rajeev Goré, Linda Postniece, and Alwen Tiu. On the correspondence between display postulates and deep inference in nested sequent calculi for tense logics. *Log. Methods Comput. Sci.*, 7(2), 2011. `doi:10.2168/LMCS-7(2:8)2011`.

11 Rajeev Goré and Revantha Ramanayake. Labelled tree sequents, tree hypersequents and nested (deep) sequents. In Thomas Bolander, Torben Braüner, Silvio Ghilardi, and Lawrence S. Moss, editors, *Advances in Modal Logic 9, papers from the ninth conference on "Advances in Modal Logic," held in Copenhagen, Denmark, 22-25 August 2012*, pages 279–299. College Publications, 2012. URL: `http://www.aiml.net/volumes/volume9/Gore-Ramanayake.pdf`.

12 Alessio Guglielmi. A system of interaction and structure. *ACM Trans. Comput. Log.*, 8(1):1, 2007. `doi:10.1145/1182613.1182614`.

13 Martin Hyland and Valeria de Paiva. Full intuitionistic linear logic (extended abstract). *Ann. Pure Appl. Log.*, 64(3):273–291, 1993. `doi:10.1016/0168-0072(93)90146-5`.

14 Nuel D. Belnap Jr. Display logic. *J. Philos. Log.*, 11(4):375–417, 1982. `doi:10.1007/BF00284976`.

15 Ryo Kashima. Cut-free sequent calculi for some tense logics. *Studia Logica*, 53(1):119–136, 1994. `doi:10.1007/BF01053026`.

16 Tim Lyon, Alwen Tiu, Rajeev Goré, and Ranald Clouston. Syntactic interpolation for tense logics and bi-intuitionistic logic via nested sequents. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, volume 152 of *LIPIcs*, pages 28:1–28:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.CSL.2020.28`.

17 Sara Negri. Proof analysis in modal logic. *J. Philos. Log.*, 34(5-6):507–544, 2005. `doi:10.1007/s10992-005-2267-3`.

18 Luís Pinto and Tarmo Uustalu. Relating sequent calculi for bi-intuitionistic propositional logic. In Steffen van Bakel, Stefano Berardi, and Ulrich Berger, editors, *Proceedings Third International Workshop on Classical Logic and Computation, CL&C 2010, Brno, Czech Republic, 21-22 August 2010*, volume 47 of *EPTCS*, pages 57–72, 2010. `doi:10.4204/EPTCS.47.7`.

19 Francesca Poggiolesi. The method of tree-hypersequents for modal propositional logic. In David Makinson, Jacek Malinowski, and Heinrich Wansing, editors, *Towards Mathematical Philosophy*, volume 28 of *Trends in logic*, pages 31–51. Springer, 2009. `doi:10.1007/978-1-4020-9084-4_3`.

20 Linda Postniece. Deep inference in bi-intuitionistic logic. In Hiroakira Ono, Makoto Kanazawa, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information and Computation, 16th International Workshop, WoLLIC 2009, Tokyo, Japan, June 21-24, 2009. Proceedings*, volume 5514 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2009. `doi:10.1007/978-3-642-02261-6_26`.

21 Alwen Tiu, Egor Ianovski, and Rajeev Goré. Grammar logics in nested sequent calculus: Proof theory and decision procedures. In Thomas Bolander, Torben Braüner, Silvio Ghilardi, and Lawrence S. Moss, editors, *Advances in Modal Logic 9, papers from the ninth conference on "Advances in Modal Logic," held in Copenhagen, Denmark, 22-25 August 2012*, pages 516–537. College Publications, 2012. URL: `http://www.aiml.net/volumes/volume9/Tiu-Ianovski-Gore.pdf`.

# A Fibrational Tale of Operational Logical Relations

**Francesco Dagnino**
University of Genova, Italy

**Francesco Gavazzo**
University of Bologna, Italy

## Abstract

Logical relations built on top of an operational semantics are one of the most successful proof methods in programming language semantics. In recent years, more and more expressive notions of operationally-based logical relations have been designed and applied to specific families of languages. However, a unifying abstract framework for operationally-based logical relations is still missing. We show how fibrations can provide a uniform treatment of operational logical relations, using as reference example a $\lambda$-calculus with generic effects endowed with a novel, abstract operational semantics defined on a large class of categories. Moreover, this abstract perspective allows us to give a solid mathematical ground also to differential logical relations – a recently introduced notion of higher-order distance between programs – both pure and effectful, bringing them back to a common picture with traditional ones.

**2012 ACM Subject Classification** Theory of computation → Operational semantics

**Keywords and phrases** logical relations, operational semantics, fibrations, generic effects, program distance

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.3

**Funding** This work was partially funded by the MUR project T-LADIES (PRIN 2020TL3X8X). The second author is supported by the ERC Consolidator Grant DLV-818616 DIAPASoN.

**Acknowledgements** The authors would like to thank the anonymous reviewers for the many useful comments, some of which improved of our work.

## 1 Introduction

Logical relations [83] are one of the most successful proof techniques in logic and programming language semantics. Introduced in proof theory [93, 43] in their unary form, logical relations have soon became a main tool in programming language semantics. In fact, starting with the seminal work by Reynolds [83], Plotkin [79], and Statman [87], logical relations have been extensively used to study both the denotational and operational behaviour of programs.[1]

Logical relations (and predicates) mostly come in two flavours, depending on whether they are defined relying on the *operational* or *denotational* semantics of a language. We refer to logical relations of the first kind as *operational logical relations* and to logical relations of the second kind as *denotational logical relations*. Due to their link with denotational semantics, denotational logical relations have been extensively studied in the last decades, both for specific programming languages and in the abstract, this way leading to beautiful general theories of (logical) predicates and relations on program denotations. In particular, starting with the work by Reynolds and Ma [68], Mitchell and Scedrov [75], and Hermida [49], researchers have started to investigate notions of (logical) predicates and relations in a

[1] See the classic textbooks by Mitchell [74], Pierce [77], and Harper [48] (and references therein) for an introduction to both denotationally- and operationally-based logical relations.

© Francesco Dagnino and Francesco Gavazzo;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 3; pp. 3:1–3:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

general categorical setting, this way giving rise to an abstract understanding of relations over (the denotational semantics of) programs centred around the notions of *fibrations* [49, 39, 86, 62, 60, 61, 59] *reflexive graphs* [76, 84, 34, 51], and *factorisation systems* [57, 54, 46]. The byproduct of all of that is a general, highly modular theory of denotational logical relations that has been successfully applied to a large array of language features, ranging from parametricty and polymorphism [39, 86, 34] to computational effects [62, 60, 61, 59, 65, 57, 46].

On the operational side, researchers have focused more on the development and applications of expressive notions of logical relations for specific (families of) languages, rather than on their underlying theory. In fact, operational logical relations being based on operational semantics, they can be easily defined for languages for which finding the right denotational model is difficult, this way making operational logical relations a handy and lightweight technique, especially when compared to its denotational counterpart. As a paradigmatic example, consider the case of stochastic $\lambda$-calculi and their operational techniques [16, 21, 97] which can be (easily) defined relying on the category of measurable spaces and measurable functions, but whose denotational semantics have required the introduction of highly non-trivial mathematical structures [89, 88, 95, 35], since the category measurable spaces (and measurable functions) is not closed [6].

The wide applicability of operational logical relations, however, has also prevented the latter to organise as a uniform *corpus* of techniques with a common underlying theory. Operational logical relations result in a mosaic of powerful techniques applied to a variety of languages – including higher-order, functional, imperative, and concurrent languages [32, 3, 94, 4, 17, 33]; both pure and (co)effectful [56, 25, 1, 12, 52, 10, 11, 13] – whose relationship, however, is unclear. This situation creates a peculiar scenario where, on the one hand, the effectiveness of operational logical relations has been proved by their many applications but, on the other hand, a foundational understanding of operational logical relations is still missing, the main consequence of that being the lack of modularity in their development. All of that becomes even worst if one also takes into account more recent forms of logical relations, such as metric [82, 78] and differential [30, 22, 23, 25] ones, that go beyond traditional relational reasoning.

In this paper, we show that much in the same way denotational logical relations can be uniformly understood in terms of fibrations, it is possible to give a uniform account of *operational logical relations* relying on the language of *fibrations*. In this respect, our contribution is twofold.

**Operational Logical Relations, Fibrationally.** Our first contribution is the development of a general, abstract notion of an operational logical relation in terms of fibrations for a $\lambda$-calculus with generic effects. Fibrations are a mainstream formalism for general, categorical notions of predicates/relations. More precisely, a fibration is a suitable functor from a category of (abstract) predicates – the domain of the fibration – to a category of arguments – usually called the base category. In denotational logical relations, predicates usually apply to program denotations rather than on program themselves, with the main consequence that the base category is usually required to be cartesian closed. In this paper, we follow a different path and work with base categories describing the operational (and interactive) behaviour of programs, rather than their denotations. To do so, we introduce the novel notion of an *operational structure*, the latter being a cartesian category with arrows describing (monadic) evaluation semantics [80, 27] and satisfying suitable coherence conditions encoding the base dynamics of program evaluation. This way, we give not only an abstract account to

traditional set-based evaluation semantics, but also of evaluation semantics going beyond the category of sets and functions, the prime example being the evaluation semantics of stochastic $\lambda$-calculi, which is defined as a stochastic kernel [21, 97].

On top of our abstract operational semantics, we give a general notion of an operational logical relation in terms of (logical) fibrations and prove a general result (which we call the fundamental lemma of logical relations, following standard nomenclature of concrete, operational logical relations) stating that programs behave as arrows in the domain of the fibration. Remarkably, our general fundamental lemma subsumes several concrete instances of the fundamental lemma of logical relations appearing in the literature. Additionally, the operational nature of our framework immediately results in a wide applicability of our results, especially if compared with fibrational accounts of denotational logical relations. In particular, since our logical relations builds upon operational structures, they can be instantiated to non-cartesian-closed categories, this way reflecting at a general level the wider applicability of operational techniques with respect to denotational ones. As a prime example of that, we obtain operational logical relations (and their fundamental lemma) for stochastic $\lambda$-calculi for free, something that is simply not achievable denotationally, due to the failure of cartesian closedness of the category of measurable spaces.

**Fibrational Differential Reasoning.** Our second, main contribution is to show that our framework goes beyond traditional relational reasoning, as it gives a novel mathematical account of the recently introduced differential logical relations [31, 22, 23, 25] (DLRs, for short), both pure and effectful. DLRs are a new form logical relations introduced to define higher-order distances between programs, such distances being abstract notions reflecting the interactive complexity of the programs compared. DLRs have been studied *operationally* and on specific calculi only, oftentimes introducing new notions – such as the one of a *differential extension* of a monad [25] – whose mathematical status is still not well understood. The main consequence of that is that a general, structural account of pure and effectful DLRs is still missing. In this paper, we show how DLRs are a specific instance of our abstract operational logical relations and how the fundamental lemma of DLRs is an instance of our general fundamental lemma. We do so by introducing the novel construction of a *fibration of differential relations* and showing how the latter precisely captures the essence of DLRs, bringing them back to a common framework with traditional logical relations. Additionally, we show how our fibrational account sheds a new light on the mathematical status of effectful DLRs. In particular, we show that differential extensions of monads are precisely liftings of monads to the fibration of differential relations, and that the so-called coupling-based differential extension [25] – whose canonicity has been left as an open problem – is an instance of a general monadic lifting to the fibration of differential relations: remarkably, such a lifting is the extension of the well-known Barr lifting [8] to a differential setting.

**Related Work.** Starting with the seminal work by Hermida [49], fibrations have been used to give categorical notions of (logical) predicates and relations, and to model denotational logical relations [39, 86, 62, 60, 61, 59], as they provide a formal way to relate the denotational semantics of a programming language and a logic for reasoning about it. Other categorical approaches to denotational logical relations have been given in terms of reflexive graphs [76, 84, 34, 51] and factorisation systems [57, 54, 46].

On the operational side, fibrations have been used to give abstract accounts to induction and coinduction [50], both in the setting of initial algebra-final coalgebra semantics [42, 41, 40] and in the setting of up-to techniques [15, 14]. To the best of the authors' knowledge,

however, none of these approaches has been applied to operational reasoning for higher-order programming languages. Concerning the latter, general operational accounts of logical relations both for effectful [56] and combined effectful and coeffectful languages [1, 26] have been given in terms of relational reasoning. These approaches, however, are tailored on specific operational semantics and notions of relations, and thus they cannot be considered truly general. Finally, DLRs have been studied mostly operationally [31, 22, 23, 25], although some general *denotational* accounts of DLRs have been proposed [31, 78]. Even if not dealing with operational aspects of DLRs, the latter proposals can cope with *pure* DLRs only, and are too restrictive to incorporate computational effects.

## 2 The Anatomy of an Operational Semantics

To define a general notion of an operational logical relation, we first need to define a general notion of an operational semantics. This is precisely the purpose of this section. In particular, we introduce the notion of an *operational structure* on a category with finite products endowed with a strong monad as an axiomatisation of a general evaluation semantics. Operational structures prescribe the existence of basic interaction arrows (the latter describing basic program interactions as given by the usual reduction rules) and define program execution as a Kleisli arrow (this way giving monadic evaluation) satisfying suitable coherence laws reflecting evaluation dynamics. Remarkably, operational structures turn out to be more liberal – hence widely applicable – than categories used in denotational semantics (the latter being required to be cartesian closed).

### 2.1 A Calculus with Generic Effects

Our target calculus is a simply-typed fine grain call-by-value [66] $\mathbf{\Lambda}$ enriched with generic effects [45, 81]. Recall that for a strong monad[2] $\mathbb{T} = (T, \eta, \gg\!=)$ on a cartesian category $\mathcal{C}$, a generic effect [81, 80] of arity $A$, with $A$ an object of $\mathcal{C}$, is an arrow $\boldsymbol{\gamma} : 1 \to TA$. Standard examples of generic effects are obtained by taking $\mathcal{C} = \mathcal{S}et$ and $A$ equal to a finite set giving the arity of the effect, so that, for instance, one can model nondeterministic (resp. fair) coins as elements of $\mathcal{P}(2)$ (resp. $\mathcal{D}(2)$), where $\mathcal{P}$ (resp. $\mathcal{D}$) is the powerset (resp. distribution) monad. Other examples of generic effects include primitives for input-output, memory updates, exceptions, etc. Here, we assume to have a collection of generic effect symbols $\gamma$ with an associated type $\sigma_\gamma$, leaving the interpretation of $\gamma$ as an actual generic effect $\boldsymbol{\gamma}$ to the operational semantics. The syntax and static semantics of $\mathbf{\Lambda}$ are given in Figure 1, where $\zeta$ ranges over base types and $c$ over constants of type $\zeta$ (for ease of exposition, we do include operations on base types, although those can be easily added).

Notice that $\mathbf{\Lambda}$'s expressions are divided into two (disjoint) classes: *values* (notation $v, w, \ldots$) and *computations* (notation $t, s, \ldots$), the former being the result of a computation, and the latter being an expression that once evaluated may produce a value (the evaluation process might not terminate) as well as side effects. When the distinction between values and computations is not relevant, we generically refer to *terms* (and still denote them as $t, s, \ldots$). We adopt standard syntactic conventions [7] and identify terms up to renaming of bound variables: we say that a term is closed if it has no free variables and write $\mathcal{V}_\sigma$, $\Lambda_\sigma$ for the sets of closed values and computations of type $\sigma$, respectively. We write $t[v_1, \ldots, v_n/x_1, \ldots, x_n]$

[2] We use the notions of a strong monad $(T, \eta, \mu, \mathsf{st})$ and of a strong Kleisli triple $\mathbb{T} = (T, \eta, \gg\!=)$ interchangeably, where for an arrow $f : X \times Y \to TZ$ in a cartesian category $\mathcal{C}$, we denote by $\gg\!= f : X \times TY \to TZ$ the strong Kleisli extension of $f$. We write $f^\dagger : TX \to TY$ for the Kleisli extension of $f : X \to TY$.

$$v, w ::= x \mid c \mid \langle\rangle \mid \lambda x.t \mid \langle v, w\rangle$$
$$t, s ::= \mathbf{val}\ v \mid vw \mid v.1 \mid v.2 \mid t\ \mathbf{to}\ x.s \mid \gamma$$

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \quad \frac{}{\Gamma \vdash c:\zeta} \quad \frac{\Gamma \vdash v:\sigma}{\Gamma \vdash \mathbf{val}\ v:\underline{\sigma}} \quad \frac{}{\Gamma \vdash \gamma:\sigma_\gamma} \quad \frac{\Gamma, x:\sigma \vdash t:\underline{\tau}}{\Gamma \vdash \lambda x.t:\sigma \to \tau} \quad \frac{\Gamma \vdash v:\sigma \to \tau \quad \Gamma \vdash w:\sigma}{\Gamma \vdash vw:\underline{\tau}}$$

$$\frac{\Gamma \vdash t:\underline{\sigma} \quad \Gamma, x:\sigma \vdash s:\underline{\tau}}{\Gamma \vdash t\ \mathbf{to}\ x.s:\underline{\tau}} \quad \frac{\Gamma \vdash v:\sigma \times \tau}{\Gamma \vdash v.1:\underline{\sigma}} \quad \frac{\Gamma \vdash v:\sigma \times \tau}{\Gamma \vdash v.2:\underline{\tau}} \quad \frac{\Gamma \vdash v:\sigma \quad \Gamma \vdash w:\tau}{\Gamma \vdash \langle v, w\rangle:\sigma \times \tau} \quad \frac{}{\Gamma \vdash \langle\rangle:\mathbf{1}}$$

**Figure 1** Syntax and Static semantics of $\mathbf{\Lambda}$.

(and similarly for values) for the capture-avoiding (simultaneous) substitution of the values $v_1, \ldots, v_n$ for all free occurrences of $x_1, \ldots, x_n$ in $t$. Oftentimes, we will use the notation $\overrightarrow{\phi}$ for a sequence $\phi_1, \ldots, \phi_n$ of symbols $\phi_i$.

When dealing with denotational logical relations, one often organises $\mathbf{\Lambda}$ as a syntactic category having types as objects and (open) terms modulo the usual $\beta\eta$-equations as arrows. Operationally, however, terms are purely syntactic objects and cannot be taken modulo $\beta\eta$-equality. For that reason, we consider a *syntactic graph* rather than a syntactic category.[3]

▶ **Definition 2.1.** *The objects of the syntactic graph $\mathcal{Syn}$ are environments $\Gamma$, types $\sigma$, and expressions $\underline{\sigma}$, for $\sigma$ a type. Arrows are defined thus:* $\hom(\Gamma, \sigma)$ *consists of values* $\Gamma \vdash v : \sigma$, *whereas* $\hom(\Gamma, \underline{\sigma})$ *consists of terms* $\Gamma \vdash t : \sigma$*; otherwise, the hom-set is empty.*

The definition of $\mathcal{Syn}$ reflects the call-by-value nature of $\mathbf{\Lambda}$: to each type $\sigma$ we associate two objects, representing the type $\sigma$ on values and on computations. Moreover, there is no arrow having environments as codomains nor having objects $\underline{\sigma}$ as domain: this reflects that in call-by-value calculi variables are placeholders for values, not for computations.

## 2.2 Operational Semantics: The Theoretical Minimum

Having defined the syntax of $\mathbf{\Lambda}$, we move to its operational semantics. Among the many style of operational semantics (small-step, big-step, etc), evaluation semantics turns out to be a convenient choice for our goals. Evaluation semantics are usually defined as monadic functions $\boldsymbol{e} : \Lambda_\sigma \to T(\mathcal{V}_\sigma)$, with $T$ a monad encoding the possible effects produced during program evaluation (e.g. divergence or nondeterminism) [27, 80, 56]. To clarify the concept, let us consider an example [27].

▶ **Example 2.2.** Let $\mathbb{T}$ be a monad on $\mathcal{Set}$ with a generic effects $\boldsymbol{\gamma} \in T(\mathcal{V}_{\sigma_\gamma})$ for effect symbols $\gamma$ in $\mathbf{\Lambda}$. The (monadic) evaluation (family of) map(s) $[\![-]\!] : \Lambda_\sigma \to T(\mathcal{V}_\sigma)$ is defined as follows (notice that $\mathbf{\Lambda}$ being simply-typed, $[\![-]\!]$ is well-defined[4]).

$$[\![\mathbf{val}\ v]\!] = \eta(v) \quad [\![(\lambda x.t)v]\!] = [\![t[v/x]]\!] \quad [\![t\ \mathbf{to}\ x.s]\!] = [\![s[\cdot/x]]\!]^\dagger [\![t]\!] \quad [\![\langle v_1, v_2\rangle.i]\!] = \eta(v_i) \quad [\![\gamma]\!] = \boldsymbol{\gamma}$$

[3] Recall that a graph is defined by removing from the definition of a category the axioms prescribing the existence of identity and composition. A diagram is a map from graphs to graphs ( a diagram being defined by removing from the definition of a functor the clauses on identity and composition). Since any category is a graph, we use the word *diagram* also to denote maps from graphs to categories.

[4] It is worth remarking that termination is not an issue for our general notion of an evaluation semantics (Definition 2.3 below). In fact, the results presented in this paper simply require to have an evaluation

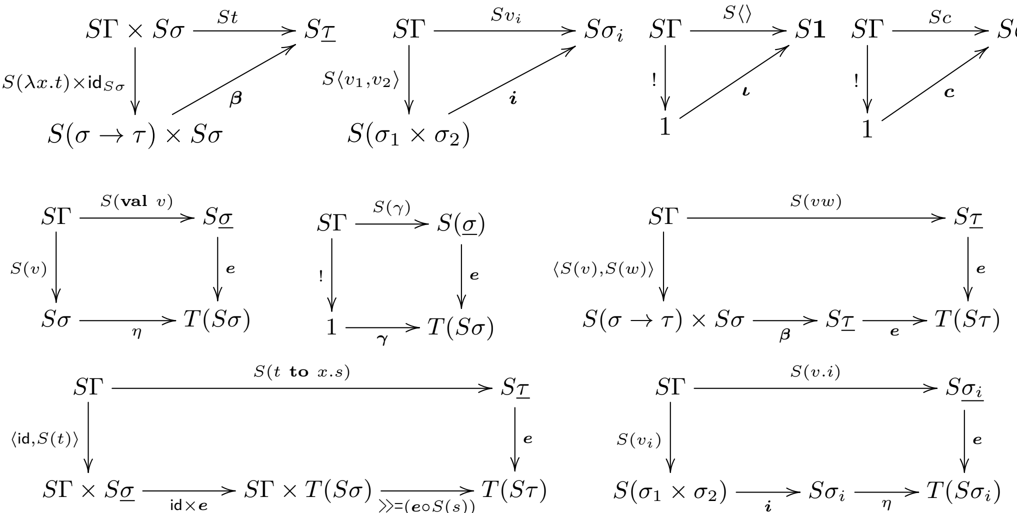


**Figure 2** Coherence laws, where $-.i \in \{-.1, -.2\}$ and $\boldsymbol{i} \in \{\boldsymbol{p_1}, \boldsymbol{p_2}\}$.

Example 2.2 defines evaluation semantics as an arrow in the category $\mathcal{Set}$ of sets and functions relying on two main ingredients: the monad $\mathbb{T}$ and its algebraic operations; and primitive functions implementing the basic mechanism of $\beta$-reductions, viz. application/substitution and projections. The very same recipe has been used to define specific evaluation semantics beyond $\mathcal{Set}$, prime example being kernel-like evaluation semantics for stochastic $\lambda$-calculi [21, 97, 95, 35], where evaluation semantics are defined as Kleisli arrows on suitable categories of measurable spaces (see Example 2.5 below for details). Here, we propose a general notion of an operational semantics for $\boldsymbol{\Lambda}$ in an arbitrary cartesian category $\mathcal{B}$ and with respect to a monad $\mathbb{T}$. We call the resulting notion a *($\mathcal{Syn}$-)operational structure.*

▶ **Definition 2.3.** *Given a category $\mathcal{B}$ with finite products and a strong monad $\mathbb{T}$ on it, a* ($\mathcal{Syn}$-)operational structure *consists of a diagram $S : \mathcal{Syn} \to \mathcal{B}$ satisfying $S(\overrightarrow{x : \sigma}) = \prod \overrightarrow{S\sigma}$, together with the following (interaction) arrows (notice that $\boldsymbol{\gamma}$ is a generic effect) and satisfying the coherence laws in Figure 2.*

$$\boldsymbol{e} : S\underline{\sigma} \to T(S\sigma) \qquad \boldsymbol{\iota} : 1 \to S\mathbf{1} \qquad \boldsymbol{\beta} : S(\sigma \to \tau) \times S\sigma \to S\underline{\tau} \qquad \boldsymbol{c} : 1 \to S\zeta$$
$$\boldsymbol{p_1} : S(\sigma \times \tau) \to S\sigma \qquad \boldsymbol{p_2} : S(\sigma \times \tau) \to S\tau \qquad \boldsymbol{\gamma} : 1 \to T(S\sigma_\gamma)$$

Notice how the first four coherence laws in Figure 2 ensure the intended behaviour of the arrows $\boldsymbol{\beta}, \boldsymbol{\iota}, \boldsymbol{p_i}, \boldsymbol{c}$, whereas the remaining laws abstractly describe the main dynamics of program execution. Notice also that Definition 2.3 prescribes the existence of an evaluation arrow $\boldsymbol{e}$: it would be interesting to find conditions on $\mathcal{B}$ (probably a domain-like enrichment [64] or partial additivity [71]) ensuring the existence of $\boldsymbol{e}$. We can now instantiate Definition 2.3 to recover standard $\mathcal{Set}$-based evaluation semantics as well as operational semantics on richer categories. In particular, since $\mathcal{B}$ need not be closed, we can give $\boldsymbol{\Lambda}$ an operational semantics in the category $\mathcal{Meas}$ of measurable spaces and measurable functions.

map satisfying suitable coherence conditions. Consequently, we could rephrase this example ignoring termination by requiring the monad to be enriched in an $\omega$-complete partial order [2] and defining evaluation semantics as a least fixed point of a suitable map, as it is customary in monadic evaluation semantics [27].

▶ **Example 2.4.** Let $\mathcal{B} = \mathcal{S}et$ and define $S : \mathcal{S}yn \to \mathcal{S}et$ thus:

$$S\sigma = \mathcal{V}_\sigma \qquad S\underline{\sigma} = \Lambda_\sigma \qquad S(\overrightarrow{x : \sigma}) = \prod \overrightarrow{S\sigma} \qquad S(\Gamma \vdash t : \sigma)(\vec{v}) = t[\vec{v}/\vec{x}]$$

We obtain an operational structure by defining the maps $\boldsymbol{\beta}, \boldsymbol{p_1}, \boldsymbol{p_2}$, and $\boldsymbol{\iota}$ in the obvious way (e.g. $\boldsymbol{\beta}(\lambda x.t, v) = t[v/x]$ and $\boldsymbol{p_i}\langle v_1, v_2\rangle = v_i$). Finally, given a monad $\mathbb{T}$ with generic effects $\boldsymbol{\gamma} \in T\mathcal{V}_{\sigma_\gamma}$ for each effect symbol $\gamma$, we define $\boldsymbol{e}$ as the evaluation map of Example 2.2.

▶ **Example 2.5** (Stochastic $\lambda$-calculus)**.** Let us consider the instance of $\boldsymbol{\Lambda}$ with a base type $\mathbf{R}$ for real numbers, constants $c_r$ for each real number $r$, and the generic effect symbol $\mathcal{U}$ standing for the uniform distribution over the unit interval. Recall that $\mathcal{M}eas$ has countable products and coproducts (but not exponentials [6]). To define the diagram $S : \mathcal{S}yn \to \mathcal{M}eas$, we rely on the well-known fact [36, 89, 16] that both $\mathcal{V}_\sigma$ and $\Lambda_\sigma$ can be endowed with a $\sigma$-algebra making them measurable (actually Polish) spaces in such a way that $\mathcal{V}_\mathbf{R} \cong \mathbb{R}$ and that the substitution map is measurable. We write $\Sigma_\sigma$ and $\Sigma_{\underline{\sigma}}$ for the $\sigma$-algebras associated to $\mathcal{V}_\sigma$ and $\Lambda_\sigma$, respectively. We thus define $S : \mathcal{S}yn \to \mathcal{M}eas$ as follows:

$$S\sigma = (\mathcal{V}_\sigma, \Sigma_\sigma) \qquad S\underline{\sigma} = (\Lambda_\sigma, \Sigma_{\underline{\sigma}}) \qquad S(\overrightarrow{x : \sigma}) = \prod \overrightarrow{S\sigma} \qquad S(\Gamma \vdash t : \sigma)(\vec{v}) = t[\vec{v}/\vec{x}]$$

We obtain an operational structure by observing that the maps $\boldsymbol{\beta}, \boldsymbol{p_1}, \boldsymbol{p_2}$, and $\boldsymbol{\iota}$ of previous example extend to $\mathcal{M}eas$, in the sense that they are all measurable functions. Next, we consider the Giry monad [44] $\mathcal{G} : \mathcal{M}eas \to \mathcal{M}eas$ which associates to each measurable space the space of probability measures on it. By Fubini-Tonelli theorem, $\mathcal{G}$ is strong. Let $\boldsymbol{\mathcal{U}}$ be the Lebesgue measure on $[0, 1]$, which we regard as an arrow $\boldsymbol{\mathcal{U}} : 1 \to \mathcal{G}(S\mathcal{V}_{\mathbf{Real}})$, and thus as a generic effect in $\mathcal{M}eas$. We then define [21] $\boldsymbol{e} : \Lambda_\sigma \to \mathcal{G}(\mathcal{V}_\sigma)$ as in Example 2.2.

## 3 Operational Logical Relations, Fibrationally

Having defined what an operational semantics for $\boldsymbol{\Lambda}$ is, we now focus on *operational* reasoning. In this section, we propose a general notion of an operational logical relation in terms of fibrations over (the underlying category of) an operational structure and prove that a general version of the fundamental lemma of logical relations holds for our operational logical relations. But before that, let us recall some preliminary notions on (bi)fibrations (we refer to [49, 55, 92] for more details).

### 3.1 Preliminaries on Fibrations

Let $p\colon \mathcal{E} \to \mathcal{B}$ be a functor and $f\colon X \to Y$ an arrow in $\mathcal{E}$ with $p(f) = u$. We say that $f$ is *cartesian* over $u$ if, for every arrow $h\colon Z \to Y$ in $\mathcal{E}$ such that $p(h) = u \circ v$, there is a unique arrow $g\colon Z \to X$ such that $p(g) = v$ and $h = f \circ g$. Dually, $f$ is *cocartesian* over $u$ if, for every arrow $h\colon X \to Z$ in $\mathcal{E}$ such that $p(h) = v \circ u$, there is a unique arrow $g\colon Y \to Z$ such that $p(g) = v$ and $h = g \circ f$. We say that $f$ is *vertical* if $u$ is an identity.

A *fibration* is a functor $p\colon \mathcal{E} \to B$ such that, for every object $X$ in $\mathcal{E}$ and every arrow $u\colon I \to p(X)$ in $\mathcal{B}$, the exists a cartesian arrow over $u$ with codomain $X$. Dually, an *opfibration* is a functor $p\colon \mathcal{E} \to \mathcal{B}$ such that for every object $X$ in $\mathcal{E}$ and every arrow $f\colon p(X) \to I$ in $\mathcal{B}$, the exists a cocartesian arrow over $u$ with domain $X$. A *bifibration* is a functor which is both a fibration and an opfibration. We refer to $\mathcal{E}$ and $\mathcal{B}$ as the domain and the base of the (bi/op)fibration. A (op)fibration is *cloven* if it comes together with a choice of (co)cartesian liftings: for an object $X$ in $\mathcal{E}$, we denote by $\overline{u}_X\colon u^*X \to X$ the

chosen cartesian arrow over $u\colon I \to p(X)$ and by $\underline{u}_X\colon X \to u_!X$ the chosen cocartesian arrow over $u\colon p(X) \to I$. A bifibration is cloven if it has choices both for cartesian and cocartesian liftings. From now on, we assume all (bi/op)fibrations to be cloven.

Let $p\colon \mathcal{E} \to B$ be a functor and $I$ an object in $\mathcal{B}$. The *fibre* over $I$ is the category $\mathcal{E}_I$ where objects are objects $X$ in $\mathcal{E}$ such that $p(X) = I$ and arrows are arrows $f\colon X \to Y$ in $\mathcal{E}$ such that $p(f) = \mathsf{id}_I$, namely, vertical arrows. Then, for every arrow $u\colon I \to J$, if $p$ is a fibration, we have a functor $u^*\colon \mathcal{E}_J \to \mathcal{E}_I$ called *reindexing along* $u$ and, if $p$ is an opfibration, we have a functor $u_!\colon \mathcal{E}_I \to \mathcal{E}_J$ called *image along* $u$. If $p$ is a bifibration, we have an adjunction $u_! \dashv u^*$.

▶ **Example 3.1.** $n$-ary predicates form a bifibration $\mathsf{Pred}_{\mathcal{S}et}\colon \mathsf{p}(\mathcal{S}et) \to \mathcal{S}et$, with $\mathsf{p}(\mathcal{S}et)$ having pairs of sets $(X, A)$ with $A \subseteq X^n$ as objects and functions $f\colon X \to Y$ such that $A \subseteq (\prod f)^{-1}(B)$ as arrows $f\colon (X, A) \to (Y, B)$, and $\mathsf{Pred}_{\mathcal{S}et}(X, A) = X$. The cartesian and cocartesian liftings of $f$ are given by inverse and direct images along $\prod f$, respectively. Special bifibrations are obtained for $n = 1$ (unary predicates) and $n = 2$ (binary relations). In those cases, we specialise the notation and write $\mathsf{Sub}_{\mathcal{S}et}\colon \mathsf{s}(\mathcal{S}et) \to \mathcal{S}et$ and $\mathsf{Rel}_{\mathcal{S}et}\colon \mathsf{r}(\mathcal{S}et) \to \mathcal{S}et$.

▶ **Example 3.2** (Weak subobjects)**.** Let $\mathcal{C}$ be a category with weak pullbacks. We define the bifibration $\Psi_{\mathcal{C}}\colon \mathsf{ws}(\mathcal{C}) \to \mathcal{C}$ of weak subobjects in $\mathcal{C}$ [47, 70]. Objects of $\mathsf{ws}(\mathcal{C})$ are pairs $(X, R)$ where $X$ is an object of $\mathcal{C}$ and $R$ is an object of the poset reflection of the slice $\mathcal{C}/X$, hence it is an equivalence class $[\alpha]$ for an arrow $\alpha\colon A \to X$ in $\mathcal{C}$. An arrow $f\colon (X, [\alpha]) \to (Y, [\beta])$ is an arrow $f\colon X \to Y$ in $\mathcal{C}$ such that $f \circ \alpha = \beta \circ g$ for some arrow $g$ in $\mathcal{C}$. Composition and identities in $\mathsf{ws}(\mathcal{C})$ are those of $\mathcal{C}$. The functor $\Psi_{\mathcal{C}}$ maps $(X, R)$ to $X$ and is the identity on arrows. It is easy to check that, for every arrow $f\colon X \to Y$ in $\mathcal{C}$ and every object $(X, [\alpha])$, the image along $f$ is $f_!(X, [\alpha]) = (Y, [f \circ \alpha])$; and for every object $(Y, [\beta])$, the reindexing along $f$ is $f^*(Y, [\beta]) = (X, [\beta'])$, where $f \circ \beta' = \beta \circ f'$ is a weak pullback square. Therefore, $\Psi_{\mathcal{C}}$ is a bifibration.

Note that, given objects $(X, [\alpha])$ and $(X, [\beta])$ in $\mathsf{ws}(\mathcal{C})$, there is at most one vertial arrow between them (the identity on $X$), hence we will write $[\alpha] \leq_X [\beta]$ when such arrow exists. Moreover, we have that $[\alpha] \leq_X [\beta]$ and $[\beta] \leq_X [\alpha]$ implies $[\alpha] = [\beta]$, by definition of poset reflection, hence the only vertical isomorphisms are identities. Finally, observe that the bifibrations $\Psi_{\mathcal{S}et}$ and $\mathsf{Sub}_{\mathcal{S}et}$ are equivalent in the sense that there is a functor $U\colon \mathsf{Sub}_{\mathcal{S}et} \to \mathsf{ws}(\mathcal{S}et)$ which is an equivalence satisfying $\Psi_{\mathcal{S}et} \circ U = \mathsf{Sub}_{\mathcal{S}et}$ and preserving (co)cocartesian arrows. The functor $U$ maps $(X, A)$ to $(X, [\iota_A])$ where $\iota_A\colon A \to X$ is the inclusion function.

Fibrations nicely carry a logical content: logical operations, in fact, can be described as categorical structures on the fibration. We now define the logical structure underlying logical relations, namely conjunctions, implications, and universal quantifiers. Recall that a fibration $p\colon \mathcal{E} \to \mathcal{B}$ has *finite products* if $\mathcal{E}$ and $\mathcal{B}$ have finite products and $p$ strictly preserves them. We denote by $\dot{\times}$ and $\dot{1}$ finite products in $\mathcal{E}$ and recall [55] that in a fibration with finite products every fibre has finite products $\wedge$ and $\top$ preserved by reindexing functors; additionally, we have the isomorphisms $A \dot{\times} B \simeq \pi_1^* A \wedge \pi_2^* B$ and $\dot{1} \simeq \top_1$.

▶ **Definition 3.3.** *A fibration $p\colon \mathcal{E} \to \mathcal{B}$ with finite products is a* logical fibration *if it is fibred cartesian closed and has universal quantifiers, where:*

1. *$p$ is* fibred cartesian closed *if every fibre $\mathcal{E}_I$ has exponentials, denoted by $X \Rightarrow Y$, and reindexings preserve them.*
2. *$p$ has* universal quantifiers *if for every projection $\pi\colon I \times J \to I$ in $\mathcal{B}$ the reindexing functor $\pi^*\colon \mathcal{E}_I \to \mathcal{E}_{I \times J}$ has a right adjoint $\forall^I_J\colon \mathcal{E}_{I \times J} \to \mathcal{E}_I$ satisfying the Beck-Chevalley condition [55].*

Note that in a fibration with universal quantifiers, we have right adjoints along any tuple of distinct projections $\langle \pi_{i_1}, \ldots, \pi_{i_k} \rangle \colon I_1 \times \ldots \times I_n \to I_{i_1} \times \ldots \times I_{i_k}$, where $i_1, \ldots, i_k \in \{1, \ldots, n\}$ are all distinct. We denote such a right adjoint by $\forall_{\langle \pi_{i_1}, \ldots, \pi_{i_k} \rangle}$. The following proposition shows under which condition the fibration of weak subobjects is a logical fibration.

▶ **Proposition 3.4** ([55, 69]). *Let $\mathcal{C}$ be a category with finite products and weak pullbacks.*

1. *If $\mathcal{C}$ is cartesian closed, then $\Psi_{\mathcal{C}}$ has universal quantifiers;*
2. *If $\mathcal{C}$ is slicewise weakly cartesian closed, then $\Psi_{\mathcal{C}}$ is fibred cartesian closed.*

▶ **Example 3.5.** Since $\mathcal{Set}$ is locally cartesian closed, Proposition 3.4 implies that both $\Psi_{\mathcal{Set}}$ and $\mathsf{Sub}_{\mathcal{Set}}$ are logical fibrations. Also $\mathsf{Rel}_{\mathcal{Set}}$ is a logical fibration, as it can be obtained from $\mathsf{Sub}_{\mathcal{Set}}$ by pulling back along the product-preserving $\mathcal{Set}$-functor $X^2 = X \times X$.

**A 2-category of (bi)fibrations.** Fibrations can be organised in a 2-category. This is important because many standard categorical concepts can be internalised in any 2-category, hence we will be able to define them also for fibrations. In particular, we will be interested in (strong) monads on a fibration, as they will allow us to define effectful logical relations. We consider the 2-category **Fib** of fibrations defined as follows. Objects are fibrations $p \colon \mathcal{E} \to \mathcal{B}$. A 1-arrow $(F, G) : p \to q$ between fibrations $p \colon \mathcal{E} \to \mathcal{B}$and $q \colon \mathcal{D} \to \mathcal{C}$is a pair of functors $F \colon \mathcal{B} \to \mathcal{C}$ and $G \colon \mathcal{E} \to \mathcal{D}$ such that $F \circ p = q \circ G$.[5] A 2-arrow $(\phi, \psi) : (F, G) \Rightarrow (H, K)$ between 1-arrows $(F, G), (H, K) : p \to q$ is a pair of natural transformations $\phi : F \dot{\to} H$ and $\psi : G \dot{\to} K$ such that $\phi p = q\psi$.. Compositions and identities are defined componentwise. The 2-category **biFib** of bifibrations is defined in the same way. We can define [90] strong monads on fibrations as strong monads in the 2-category **Fib**. That is, a monad $\mathbb{T}$ on a fibration $p \colon \mathcal{E} \to \mathcal{B}$ consists of the following data: a 1-arrow $(T, S) : p \to p$ and two 2-arrows $(\mu_T, \mu_S) : (T^2, S^2) \Rightarrow (T, S)$, $(\eta_T, \eta_S) : (\mathsf{Id}_{\mathcal{E}}, \mathsf{Id}_{\mathcal{B}}) \Rightarrow (T, S)$, and $(\mathsf{st}_T, \mathsf{st}_S) : (- \times T-, - \dot{\times} S-) \Rightarrow (T(- \times -), S(- \dot{\times} -))$ such that $(T, \mu_T, \eta_T, \mathsf{st}_T)$ is a strong monad on $\mathcal{B}$ and $(S, \mu_S, \eta_S, \mathsf{st}_S)$ is a strong monad on $\mathcal{E}$. In particular, given a strong monad $\mathbb{T} = (T, \mu, \eta, \mathsf{st})$ on $\mathcal{B}$, a lifting of $\mathbb{T}$ to $p \colon \mathcal{E} \to \mathcal{B}$ is a tuple $\dot{\mathbb{T}} = (\dot{T}, \dot{\mu}, \dot{\eta}, \dot{\mathsf{st}})$ such that $(T, \dot{T}, \mu, \dot{\mu}, \eta, \dot{\eta}, \mathsf{st}, \dot{\mathsf{st}})$ is a strong monad on $p \colon \mathcal{E} \to \mathcal{B}$. We write $\dot{\gg=}$ for the lifting of $\gg=$.

## 3.2 Operational Logical Relations and Their Fundamental Lemma

We are now ready to define a general notion of an operational logical relation. Let us consider an operational structure over a cartesian category $\mathcal{B}$ with a strong monad $\mathbb{T}$ on $\mathcal{B}$, as in Definition 2.3. Let $p \colon \mathcal{E} \to \mathcal{B}$ be a logical fibration and $\dot{\mathbb{T}}$ be a lifting of $\mathbb{T}$ to $p$.

▶ **Definition 3.6.** *A logical relation is a mapping $R$ from objects of $\mathcal{Syn}$ to objects of $\mathcal{E}$ such that $p(Rx) = Sx$, for any object $x$ of $\mathcal{Syn}$,*[6] *and the following hold.*

$$R\mathbf{1} = \top_{S\mathbf{1}} \qquad R(\sigma \times \tau) = \boldsymbol{p}_1^*(R\sigma) \wedge \boldsymbol{p}_2^*(R\tau) \qquad R\underline{\sigma} = \boldsymbol{e}^*(\dot{T}(R\sigma))$$

$$R(\overrightarrow{x : \sigma}) = \dot{\prod} \overrightarrow{R\sigma} \qquad R(\sigma \to \underline{\tau}) = \forall_{\pi_1} \pi_2^*(R\sigma) \Rightarrow \boldsymbol{\beta}^*(R\underline{\tau})$$

Notice that giving a logical relation essentially amounts to specify the action of $R$ on basic types, since the action of $R$ on complex types is given by Definition 3.6. The defining clauses of a logical relation exploit both the logic of a (logical) fibration and the operational

[5] Note that we do not require $G$ to preserve cartesian arrows.

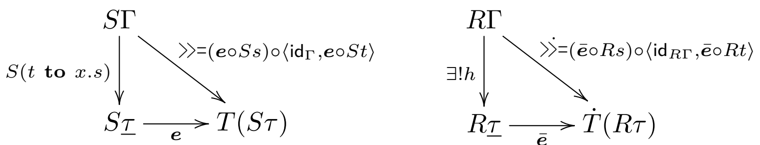
[6] Actually, it suffices to have $p(R\zeta) = S\zeta$ for basic types only, as the second part of the definition assures it for other types.

semantics of $\mathbf{\Lambda}$. The reader should have recognised in Definition 3.6 the usual definition of a logical relation, properly generalised to rely on the logic of a fibration only. For instance, $R\underline{\sigma}$ intuitively relates computations whose evaluations are related by the lifting of the monad. Notice also that the clause of arrow types has a higher logical complexity than other clauses, as it involves *two* logical connectives, viz. implication and universal quantification.

Operational logical relations come with their so-called *fundamental lemma*, which states that (open) terms maps (via substitution) related values to related terms. In our abstract framework that to any term $t$ we can associate a suitable arrow $Rt$ in $\mathcal{E}$ lying above $St$. To prove our general version of the fundamental theorem, we have to assume it for the parameters of our calculus (constants of basic types and generic effects). Accordingly, we say that a logical relation $R$ is ($\mathbf{\Lambda}$-)*stable* if: (i) for every constant $c$ of a base type $\zeta$, we have an arrow $\dot{\boldsymbol{c}} : \dot{1} \to R\zeta$ above $\boldsymbol{c}$; (ii) we have an arrow $\dot{\boldsymbol{\gamma}} : \dot{1} \to \dot{T}(R\sigma)$ above $\boldsymbol{\gamma}$.

▶ **Theorem 3.7** (Fundamental Lemma). *Let $R$ be a stable logical relation. The map $R$ extends to a diagram $R : \mathcal{Syn} \to \mathcal{E}$ such that $p \circ R = S$. In particular, for any term $\Gamma \vdash t : \sigma$, there is an arrow $Rt : R\Gamma \to R\underline{\sigma}$ in $\mathcal{E}$ above $St$ (similarly, for values).*

**Proof sketch.** Given $\Gamma \vdash t : \sigma$, we construct the desired arrow $Rt$ by induction on $t$. The case for values lifts commutative triangles in Figure 2 using the universal property of cartesian liftings of interaction arrows and then constructs the desired arrow using the logical structure of $p$ and $R\sigma$. The case of terms, just lifts the commutative diagrams in Figure 2 using the universal property of the cartesian lifting of the $\boldsymbol{e}$. As a paradigmatic example, we show the case of sequencing. First, let us notice that for any type $\sigma$, the evaluation arrow $\boldsymbol{e} : S\underline{\sigma} \to T(S\sigma)$ gives a cartesian arrow $\bar{\boldsymbol{e}} : \boldsymbol{e}^*(\dot{T}(R\sigma)) \to \dot{T}(R\sigma)$, i.e. $\bar{\boldsymbol{e}} : R\underline{\sigma} \to \dot{T}(R\sigma)$ (since $R\underline{\sigma} = \boldsymbol{e}^*(\dot{T}(R\sigma))$). Let us now consider the case of $\Gamma \vdash t\ \mathbf{to}\ x.s : \underline{\tau}$ as obtained from $\Gamma \vdash t : \sigma$ and $\Gamma, x : \sigma \vdash s : \tau$. By induction hypothesis, we have arrows $Rt : R\Gamma \to R\underline{\sigma}$ and $Rs : R\Gamma \dot{\times} R\sigma \to R\underline{\tau}$. By postcomposing the former with $\bar{\boldsymbol{e}}$, we obtain the arrow $\bar{\boldsymbol{e}} \circ Rt : R\Gamma \to \dot{T}(R\sigma)$, and thus $\langle \mathsf{id}_{R\Gamma}, \bar{\boldsymbol{e}} \circ Rt\rangle : R\Gamma \to R\Gamma \dot{\times} \dot{T}(R\sigma)$. In a similar fashion, we have $\bar{\boldsymbol{e}} \circ Rs : R\Gamma \dot{\times} R\sigma \to \dot{T}(R\tau)$ and thus, using the extension of the monad, $\dot{\gg\!=}(\bar{\boldsymbol{e}} \circ Rs) : R\Gamma \dot{\times} \dot{T}(R\sigma) \to \dot{T}(R\tau)$. Altogether, we obtain the arrow $\dot{\gg\!=}(\bar{\boldsymbol{e}} \circ Rs) \circ \langle \mathsf{id}_{R\Gamma}, \bar{\boldsymbol{e}} \circ Rt\rangle : R\Gamma \to \dot{T}(R\tau)$. Using the commutative diagram of sequencing in Figure 2 and the very definition of a fibration, we obtain




We choose $h$ as $R(t\ \mathbf{to}\ x.s)$. ◀

We can now instantiate Theorem 3.7 with the operational structures and fibrations seen so far to recover traditional logical relations (and their fundamental lemmas). For instance, the operational structure of Example 2.5 and the fibraration obtained by pulling back $\mathsf{Rel}_{\mathcal{Set}}$ along the forgetful from $\mathcal{Meas}$ to $\mathcal{Set}$ (together with a lifting of the Giry monad [62, 60]) give operational logical relations for stochastic $\lambda$-calculi. Theorem 3.7 then gives compositionality (i.e. congruence and substitutivity) of the logical relation. But that is not the end of the story. In fact, our general results go beyond the realm of traditional logical relations.

# 4 The fibration of differential relations

In this section, we describe the construction of the *fibration of differential relations*, which can serve as a fibrational foundation of *differential logical relations* [31, 22, 23, 25] (DLRs), a recently introduced form of logical relations defining higher-order distances between programs. DLRs are ternary relations relating pairs of terms with elements representing distances between them: such distances, however, need not be numbers. More precisely, with each type $\sigma$ one associates a set $(\!|\sigma|\!)$ of (higher-order) distances between terms of type $\sigma$, and then defines DLRs as relating terms of type $\sigma$ with distances in $(\!|\sigma|\!)$ between them. Elements of $(\!|\sigma|\!)$ reflect the interactive complexity of programs, the latter being given by the type $\sigma$. Thus, for instance, the main novelty of DLRs is that a distance between two values $\lambda x.t, \lambda x.s$ of type $\sigma \to \tau$ is not *just* a number, but a function $dt : \mathcal{V}_\sigma \times (\!|\sigma|\!) \to (\!|\tau|\!)$ mapping a value $v$ and an error/perturbation $dv$ to an error/perturbation $dt(v, dv)$. A DLR then relates $\lambda x.t$ and $\lambda x.s$ to $dt$ if for all values $v, w$ related to $dv$ (meaning that $v$ and $w$ are $dv$-apart), then $t[v/x]$ and $s[w/x]$ are related to $dt(v, dv)$.

Semantically, DLRs give rise to generalised distance spaces and differential extensions of monads, the former being relational structures $(X, (\!|X|\!), \delta_X)$ with $\delta_X \subseteq X \times (\!|X|\!) \times X$ acting as the semantic counterpart of a DLR, and the latter being reminiscent[7] of extensions of monads to the category of generalised distance spaces. Here, we show how our general notion of an operational logical relation subsumes the one of a DLR (and, consequently, that the fundamental lemma of DLRs is an instance of Theorem 3.7). Additionally, we show how differential extensions are precisely liftings of monads to the fibration of differential relations and how the so-called coupling-based differential extension [25] is an instance of a general monadic lifting to such a fibration, viz. the well-known Barr lifting properly fitted to a differential setting.

## 4.1 Going Differential, Fibrationally

Let $p\colon \mathcal{E} \to \mathcal{B}$ be a fibration with finite products.[8] We define the category $\mathsf{dr}(p)$ of *differential relations* in $p$ as follows:

**Objects** are triples $\mathsf{X} = (|\mathsf{X}|, (\!|\mathsf{X}|\!), \delta_\mathsf{X})$, where $|\mathsf{X}|$ and $(\!|\mathsf{X}|\!)$ are objects in $\mathcal{B}$ and $\delta_\mathsf{X}$ is an object in the fibre $\mathcal{E}_{|\mathsf{X}| \times (\!|\mathsf{X}|\!) \times |\mathsf{X}|}$.

**Arrows** $f\colon \mathsf{X} \to \mathsf{Y}$ are triples $f = (|f|, \mathsf{d}f, \varphi_f)$, where $|f|\colon |\mathsf{X}| \to |\mathsf{Y}|$ and $\mathsf{d}f\colon |\mathsf{X}| \times (\!|\mathsf{X}|\!) \to (\!|\mathsf{Y}|\!)$ are arrows in $\mathcal{B}$, and $\varphi_f\colon \delta_\mathsf{X} \to \delta_\mathsf{Y}$ is an arrow in $\mathcal{E}$ over $\langle |f|\pi_1, \mathsf{d}f\langle \pi_1, \pi_2\rangle, |f|\pi_3\rangle$.

**Composition** of arrows $f\colon \mathsf{X} \to \mathsf{Y}$ and $g\colon \mathsf{Y} \to \mathsf{Z}$ is defined thus:

$$|g \circ f| = |g| \circ |f| \qquad \mathsf{d}(g \circ f) = \mathsf{d}g \circ \langle |f|\pi_1, \mathsf{d}f\rangle \qquad \varphi_{g\circ f} = \varphi_g \circ \varphi_f.$$

**Identity** on $\mathsf{X}$ is given by $\mathsf{id}_\mathsf{X} = (\mathsf{id}_{|\mathsf{X}|}, \pi_2, \mathsf{id}_{\delta_\mathsf{X}})$.

▶ **Proposition 4.1.** $\mathsf{dr}(p)$ *is a category with finite products.*

▶ Remark 4.2. Note that an arrow $f\colon \mathsf{X} \to \mathsf{Y}$ in $\mathsf{dr}(p)$ can be equivalently described as a triple $(|f|, \mathsf{D}f, \varphi_f)$ where $|f|$ is as before, $\mathsf{D}f\colon |\mathsf{X}| \times (\!|\mathsf{X}|\!) \to |\mathsf{Y}| \times (\!|\mathsf{Y}|\!)$ is such that $|f| \circ \pi_1 = \pi_1 \circ \mathsf{D}f$ and $\varphi_f\colon \delta_\mathsf{X} \to \delta_\mathsf{Y}$ is above $\mathsf{D}f \times |f|\colon |\mathsf{X}| \times (\!|\mathsf{X}|\!) \times |\mathsf{X}| \to |\mathsf{Y}| \times (\!|\mathsf{Y}|\!) \times |\mathsf{Y}|$. This presentation is perhaps more in the spirit of fibrations, but we opted for the other one, which follows the original presentation of generalised distance spaces [25].

[7] Whether differential extensions indeed define a monadic lifting is left as an open problem in [25].

[8] Actually, to carry out our construction, binary products in the base are enough, but we use a richer structure as we need it in the following part of this paper.

In the following, we denote by $\nabla\mathsf{X}$ the object $|\mathsf{X}| \times (\!|\mathsf{X}|\!) \times |\mathsf{X}|$ of $\mathcal{B}$ and by $\nabla f$ the arrow $\langle |f|\pi_1, \mathsf{d}f\langle \pi_1, \pi_2\rangle, |f|\pi_3\rangle$. These data define a functor $\nabla\colon \mathsf{dr}(p) \to \mathcal{B}$.

▶ **Example 4.3.** For the fibration $\mathsf{Sub}_{\mathcal{S}et}\colon \mathsf{s}(\mathcal{S}et) \to \mathcal{S}et$ of Example 3.1, the category $\mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et})$ is the category of generalised distance spaces [25]. An object in $\mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et})$ is essentially a triple $(X, V, R)$ consisting of a set $X$ of points, a set $V$ of distance values, and a ternary relation $R \subseteq X \times V \times X$ specifying at which distance two elements of $X$ are related: that is, $(x, v, y) \in R$ means that $x$ and $y$ are related at distance $v$. For instance, a metric $d\colon X \times X \to [0, \infty]$ on $X$ can be seen as a ternary relation $R_d \subseteq X \times [0, \infty] \times X$ defined by $(x, v, y) \in R_d$ iff $d(x, y) \leq v$. An arrow from $(X, V, R)$ to $(Y, U, S)$ in $\mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et})$ consists of a function $|f|\colon X \to Y$ transforming points together with a function $\mathsf{d}f\colon X \times V \to U$ transforming distance values such that, for all $x, y \in X$ and $v \in V$, $(x, v, y) \in R$ implies $(|f|(x), \mathsf{d}f(x, v), |f|(y)) \in S$.

The assignments $\mathsf{DRel}^p(\mathsf{X}) = |\mathsf{X}|$ and $\mathsf{DRel}^p(f) = |f|$ determine a functor $\mathsf{DRel}^p\colon \mathsf{dr}(p) \to \mathcal{B}$.

▶ **Proposition 4.4.** *The functor* $\mathsf{DRel}^p\colon \mathsf{dr}(p) \to \mathcal{B}$ *is a fibration with finite products.*

**Proof.** Define the cartesian lifting along an arrow $u\colon I \to J$ of an object $\mathsf{Y}$ with $|\mathsf{Y}| = J$ by $(u, \pi_2, \overline{\langle u\pi_1, \pi_2, u\pi_3\rangle}_{\delta_\mathsf{Y}})\colon u^*\mathsf{Y} \to \mathsf{Y}$, where $u^*\mathsf{Y} = (I, (\!|\mathsf{Y}|\!), \langle u\pi_1, \pi_2, u\pi_3\rangle^*\delta_\mathsf{Y})$. ◀

▶ Remark 4.5. The fibration $\mathsf{DRel}^p$ can be also obtained from the *simple fibration* [55] $s\colon \mathsf{s}(\mathcal{B}) \to \mathcal{B}$, where objects of $\mathsf{s}(\mathcal{B})$ are pairs $(I, X)$ of objects in $\mathcal{B}$; arrows $(u, f)\colon (I, X) \to (J, Y)$ are pairs of arrows $u\colon I \to J$ and $f\colon I \times X \to Y$ in $\mathcal{B}$; and $s$ projects both objects and arrows on the first component. Indeed, we have that $\mathsf{DRel}^p = s \circ p'$ where $p'\colon \mathsf{dr}(p) \to \mathsf{s}(\mathcal{B})$ is obtained by the pullback

$$\begin{array}{ccc} \mathsf{dr}(p) & \longrightarrow & \mathcal{E} \\ {\scriptstyle p'}\downarrow & \scriptstyle p.b. & \downarrow{\scriptstyle p} \\ \mathsf{s}(\mathcal{B}) & \longrightarrow & \mathcal{B} \end{array}$$

, with the functor $\mathsf{s}(\mathcal{B}) \to \mathcal{B}$ mapping $(I, X)$ to $I \times X \times I$ and $(u, f)$ to $\langle u\pi_1, f\langle \pi_1, \pi_2\rangle, u\pi_3\rangle$. Therefore, we have $p'(\mathsf{X}) = (|\mathsf{X}|, (\!|\mathsf{X}|\!))$ and $p'(f) = (|f|, \mathsf{d}f)$. This is somewhat similar to the simple coproduct completion of a fibration [53]. We leave a precise comparison between these constructions for future work.

We now show under which conditions the fibration $\mathsf{DRel}^p$ is a logical fibration.

▶ **Proposition 4.6.** *If* $p\colon \mathcal{E} \to \mathcal{B}$ *is a logical fibration and* $\mathcal{B}$ *is cartesian closed, then* $\mathsf{DRel}^p$ *is a logical fibration.*

We report the definition of the logical structure on $\mathsf{DRel}^p$: let $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ objects in $\mathsf{dr}(p)$ with $|\mathsf{X}| = |\mathsf{Y}| = I$ and $|\mathsf{Z}| = I \times J$.

$$\widehat{\top}_I = (I, 1, \top_{I\times 1\times I}) \qquad \mathsf{X}\widehat{\wedge}_I\mathsf{Y} = (I, (\!|\mathsf{X}|\!) \times (\!|\mathsf{Y}|\!), \langle \pi_1, \pi_2, \pi_4\rangle^*\delta_\mathsf{X} \wedge \langle \pi_1, \pi_3, \pi_4\rangle^*\delta_\mathsf{Y})$$

$$\mathsf{X}\widehat{\Rightarrow}_I\mathsf{Y} = (I, [(\!|\mathsf{X}|\!), (\!|\mathsf{Y}|\!)], \forall_{\langle \pi_1,\pi_2,\pi_4\rangle}(\langle \pi_1, \pi_3, \pi_4\rangle^*\delta_\mathsf{X} \Rightarrow \langle \pi_1, \mathsf{ev}^{(\!|\mathsf{X}|\!)}_{(\!|\mathsf{Y}|\!)}\langle \pi_2, \pi_3\rangle, \pi_4\rangle^*\delta_\mathsf{Y}))$$

$$\widehat{\forall}^I_J\mathsf{Z} = (I, [J, (\!|\mathsf{Z}|\!)], \forall_{\langle \pi_1,\pi_3,\pi_5\rangle}(\langle \pi_1, \pi_2, \mathsf{ev}^J_{(\!|\mathsf{Z}|\!)}\langle \pi_3, \pi_2\rangle, \pi_4, \pi_5\rangle^*\delta_\mathsf{Z}))$$

▶ **Example 4.7.** Let us consider the fibration $\mathsf{DRel}^{\mathsf{Sub}_{\mathcal{S}et}}\colon \mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et}) \to \mathcal{S}et$ and instantiate the above constructions to it. We have $\widehat{\top}_X = (X, 1, X \times 1 \times X)$, that is, in $\widehat{\top}_X$ there is just one distance value and all elements of $X$ are related. Consider now objects $\mathsf{X} = (X, V, R)$ and $\mathsf{Y} = (X, U, S)$ in $\mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et})$. Then, we have $\mathsf{X}\widehat{\wedge}\mathsf{Y} = (X, V \times U, R \sqcap S)$ and $\mathsf{X}\widehat{\Rightarrow}\mathsf{Y} = (X, [V, U], R \to S)$, where

- $(x,(v,u),y) \in R \sqcap S$ iff $(x,v,y) \in R$ and $(x,u,y) \in S$,
- $(x,f,y) \in R \to S$ iff, for all $v \in V$, $(x,v,y) \in R$ implies $(x,f(v),y) \in S$.

That is, two elements $x$ and $y$ are related in $R \sqcap S$ at a distance $(v,u)$ such that $x$ and $y$ are at distance $v$ in $R$ and $u$ in $S$. Instead, $x$ and $y$ are related in $R \to S$ at a distance $f$ transforming distances in $V$ into distances in $U$, respecting $R$ and $S$. Finally, if $\mathsf{Z} = (X \times Y, V, R)$ is an object in $\mathsf{dr}(\mathsf{Sub}_{\mathcal{C}}\mathcal{S}et])$, then we have $\widehat{\forall}^X_Y \mathsf{Z} = (X, [Y,V], Y \to R)$, where $(x,f,x') \in Y \to R$, iff for all $y,y' \in Y$, $((x,y),f(y),(x',y')) \in R$. That is, elements $x$ and $x'$ are related by $Y \to R$ at a distance $f$ returning for each $y \in Y$ a distance in $V$ such that $R$ relates $(x,y)$ and $(x',y')$ at distance $f(y)$, for each $y,y' \in Y$.

We now extend this construction to 1- and 2-arrows. To do so, we work with bifibrations whose bases have finite products. Let $p$ and $q$ be such bifibrations and $(F,G) : p \to q$ be a 1-arrow in **biFib**. We define a functor $\widehat{G} \colon \mathsf{dr}(p) \to \mathsf{dr}(q)$ thus: for $\mathsf{X}$ an object in $\mathsf{dr}(p)$, we set

$$|\widehat{G}\mathsf{X}| = F|\mathsf{X}| \qquad (\!|\widehat{G}\mathsf{X}|\!) = F(|\mathsf{X}| \times (\!|\mathsf{X}|\!)) \qquad \delta_{\widehat{G}\mathsf{X}} = \mathrm{pr}^F_{\mathsf{X}\,!} G\delta_{\mathsf{X}},$$

where $\mathrm{pr}^F_{\mathsf{X}} = \langle F\pi_1, F\langle \pi_1,\pi_2\rangle, F\pi_3\rangle \colon F(|\mathsf{X}| \times (\!|\mathsf{X}|\!) \times |\mathsf{X}|) \to F|\mathsf{X}| \times F(|\mathsf{X}| \times (\!|\mathsf{X}|\!)) \times F|\mathsf{X}|$ is an arrow in $\mathcal{B}$. For every arrow $f \colon \mathsf{X} \to \mathsf{Y}$ in $\mathsf{dr}(p)$, we set

$$|\widehat{G}f| = F|f| \qquad \mathsf{d}(\widehat{G}f) = (F\langle |f|\pi_1, \mathsf{d}f\rangle)\pi_2 \qquad \varphi_{\widehat{G}f} = \upsilon(G\varphi_f),$$

where $\upsilon(G\varphi_f)$ is the unique arrow over $F|f| \times F\langle |f|\pi_1, \mathsf{d}f\rangle \times F|f|$ making the diagram on the left commute, which exists as the diagram on the right commutes and $\underline{\mathrm{pr}^F_{\mathsf{X}}}_{G\delta_{\mathsf{X}}}$ is cocartesian.

$$\begin{array}{ccc} G\delta_{\mathsf{X}} & \xrightarrow{\underline{\mathrm{pr}^F_{\mathsf{X}}}_{G\delta_{\mathsf{X}}}} & \delta_{\widehat{G}\mathsf{X}} \\ {\scriptstyle G\varphi_f}\downarrow & & \vdots\,{\scriptstyle \upsilon(G\varphi_f)} \\ G\delta_{\mathsf{Y}} & \xrightarrow[\underline{\mathrm{pr}^F_{\mathsf{Y}}}_{G\delta_{\mathsf{Y}}}]{} & \delta_{\widehat{G}\mathsf{Y}} \end{array} \qquad \begin{array}{ccc} F(\nabla\mathsf{X}) & \xrightarrow{\mathrm{pr}^F_{\mathsf{X}}} & F|\mathsf{X}| \times F(|\mathsf{X}| \times (\!|\mathsf{X}|\!)) \times F|\mathsf{X}| \\ {\scriptstyle F(\nabla f)}\downarrow & & \downarrow{\scriptstyle F|f| \times F\langle |f|\pi_1, \mathsf{d}f\rangle \times F|f|} \\ F(\nabla\mathsf{Y}) & \xrightarrow[\mathrm{pr}^F_{\mathsf{Y}}]{} & F|\mathsf{Y}| \times F(|\mathsf{Y}| \times (\!|\mathsf{Y}|\!)) \times F|\mathsf{Y}| \end{array}$$

Notice that cocartesian liftings are essential to appropriately define $\widehat{G}$ on the relational part of $\mathsf{X}$ and $f$, as we do not assume any compatibility with products for $F$.

▶ **Proposition 4.8.** $(F,\widehat{G}) : \mathsf{DRel}^p \to \mathsf{DRel}^q$ *is a 1-arrow in* **Fib***.*

Similarly, given a 2-arrow $(\phi,\psi) : (F,G) \Rightarrow (H,K)$ in **biFib** between 1-arrows $(F,G) : p \to q$ and $(H,K) : p \to q$, we define a natural transformation $\widehat{\psi} : \widehat{G} \xrightarrow{\cdot} \widehat{K}$ as follows: for every object $\mathsf{X}$ in $\mathsf{DRel}^p$, we set:

$$|\widehat{\psi}_{\mathsf{X}}| = \phi_{|\mathsf{X}|} \qquad \mathsf{d}(\widehat{\psi}_{\mathsf{X}}) = \phi_{|\mathsf{X}| \times (\!|\mathsf{X}|\!)}\pi_2 \qquad \varphi_{\widehat{\psi}_{\mathsf{X}}} = \upsilon(\psi_{\delta_{\mathsf{X}}}),$$

where $\upsilon(\psi_{\delta_{\mathsf{X}}})$ is the unique arrow over $\phi_{|\mathsf{X}|} \times \phi_{|\mathsf{X}| \times (\!|\mathsf{X}|\!)} \times \phi_{|\mathsf{X}|}$ making the diagram on the left commute, which exists as the diagram on the right commutes and $\underline{\mathrm{pr}^F_{\mathsf{X}}}_{G\delta_{\mathsf{X}}}$ is cocartesian.

$$\begin{array}{ccc} G\delta_{\mathsf{X}} & \xrightarrow{\underline{\mathrm{pr}^F_{\mathsf{X}}}_{G\delta_{\mathsf{X}}}} & \delta_{\widehat{G}\mathsf{X}} \\ {\scriptstyle \psi_{\delta_{\mathsf{X}}}}\downarrow & & \vdots\,{\scriptstyle \upsilon(\psi_{\delta_{\mathsf{X}}})} \\ K\delta_{\mathsf{X}} & \xrightarrow[\underline{\mathrm{pr}^H_{\mathsf{X}}}_{K\delta_{\mathsf{X}}}]{} & \delta_{\widehat{K}\mathsf{X}} \end{array} \qquad \begin{array}{ccc} F(\nabla\mathsf{X}) & \xrightarrow{\mathrm{pr}^F_{\mathsf{X}}} & F|\mathsf{X}| \times F(|\mathsf{X}| \times (\!|\mathsf{X}|\!)) \times F|\mathsf{X}| \\ {\scriptstyle \phi_{\nabla\mathsf{X}}}\downarrow & & \downarrow{\scriptstyle \phi_{|\mathsf{X}|} \times \phi_{|\mathsf{X}| \times (\!|\mathsf{X}|\!)} \times \phi_{|\mathsf{X}|}} \\ H(\nabla\mathsf{X}) & \xrightarrow[\mathrm{pr}^H_{\mathsf{X}}]{} & H|\mathsf{X}| \times H(|\mathsf{X}| \times (\!|\mathsf{X}|\!)) \times H|\mathsf{X}| \end{array}$$

▶ **Proposition 4.9.** $(\phi, \widehat{\psi}) : (F, \widehat{G}) \Rightarrow (H, \widehat{K})$ *is a 2-arrow in* $\mathbf{Fib}$*.*

We are getting closer to the definition of a 2-functor DR from (certain) bifibrations to fibrations given by the following assignments:

$$\mathrm{DR}(p) = \mathsf{DRel}^p \qquad \mathrm{DR}(F, G) = (F, \widehat{G}) \qquad \mathrm{DR}(\phi, \psi) = (\psi, \widehat{\psi}).$$

However, this is not the case as DR does not preserve identity 1-arrows. Indeed, given the identity $(\mathsf{Id}_{\mathcal{B}}, \mathsf{Id}_{\mathcal{E}})$ on a bifibration $p\colon \mathcal{E} \to \mathcal{B}$, we have that for every object $\mathsf{X}$ in $\mathsf{dr}(p)$, $\widehat{\mathsf{Id}}_{\mathcal{E}}(\mathsf{X}) = (|\mathsf{X}|, |\mathsf{X}| \times (\!|\mathsf{X}|\!), \mathrm{pr}_{\mathsf{X}}^{\mathsf{Id}_{\mathcal{B}}}{}_{!}\delta_{\mathsf{X}})$, which is not isomorphic to $\mathsf{X}$, in general.

We can recover a form of functoriality by restricting to a 2-subcategory of $\mathbf{biFib}$, notably, considering only 1-arrows which are cocartesian. We say that a 1-arrow $(F, G) : p \to q$ in $\mathbf{biFib}$ is *cocartesian* if the functor $G$ preserves cocartesian arrows. This implies that $G$ commutes with cocartesian liftings up to isomorphism. Denote by $\mathbf{biFib}_c$ the 2-subcategory of $\mathbf{biFib}$ where objects have finite products and 1-arrows are cocartesian.

▶ **Theorem 4.10.** $DR\colon \mathbf{biFib}_c \to \mathbf{Fib}$ *is a lax functor.*

Note that we just have a *lax functor*, as identities and composition of 1-arrows are preserved only up to a family of a mediating 2-arrow. Nonetheless, this is enough to get important properties of the construction.

## 4.2 Lifting monads: the differential extension

An important problem when dealing with effectful languages is the lifting of monads from the category where the semantics of the language is expressed to the category used to reason about programs, e.g., the domain of a fibration or a category of relations. The most famous one is perhaps the so-called *Barr extension* [8] of a $\mathcal{S}et$-monad to the category of (endo)relations, which is fibred over $\mathcal{S}et$ (other notions of lifting include $\top\top$- and codensity lifting [62, 60]). In the differential setting, the notion of a *differential extension* has been recently proposed [25] as a way to lift monads to generalised distance spaces. To what extent such a construction is canonical and whether it defines an actual monadic lifting, however, have been left as open questions. Here, we answer both questions in the affirmative.

▶ **Definition 4.11.** *Let* $p\colon \mathcal{E} \to \mathcal{B}$ *be a fibration with finite products and* $\mathbb{T} = (T, \mu, \eta, \mathsf{st})$ *be a strong monad on* $\mathcal{B}$*. A* differential extension *of* $\mathbb{T}$ *along* $p$ *is a lifting* $\dot{\mathbb{T}} = (\dot{T}, \dot{\mu}, \dot{\eta}, \dot{\mathsf{st}})$ *of* $\mathbb{T}$ *along the fibration* $\mathsf{DRel}^p\colon \mathsf{dr}(p) \to \mathcal{B}$*.*

A differential extension of $\mathbb{T}$ along the fibration $p$ is thus a monad on the category of differential relations in $p$ which is above $\mathbb{T}$ with respect to the fibration $\mathsf{DRel}^p$. We describe two techniques to build differential extensions of monads. The first one is an immediate consequence of Theorem 4.10. Indeed, $\mathrm{DR}\colon \mathbf{biFib}_c \to \mathbf{Fib}$ being a lax functor, every cocartesian monad on a bifibration $p$ induces a monad on the fibration $\mathsf{DRel}^p$, hence a differential extension. This follows from general properties of lax functors [9, 91].

▶ **Corollary 4.12.** *Let* $\mathbb{T} = (T, S, \eta_T, \eta_S, \mu_T, \mu_S)$ *be a monad on* $p$ *in* $\mathbf{biFib}_c$*. Then,* $\widehat{\mathbb{T}} = (T, \widehat{S}, \eta_T, \widehat{\eta_S}, \mu_T, \widehat{\mu_S})$ *is a monad on* $\mathsf{DRel}^p$ *in* $\mathbf{Fib}$*.*

To get a lifting of strong monads, we need an additional hypothesis: we have to require that the product functor $\dot{\times}$ on the total category $\mathcal{E}$ preserves cocartesian arrows, as often happens when dealing with monoidal bifibrations [85, 73].[9] This is sensible as the action of DR on 1-arrows is defined using cocartesian liftings.

[9] Here the monoidal structure is given just by cartesian product.

▶ **Theorem 4.13.** *Let $p\colon \mathcal{E} \to \mathcal{B}$ be a bifibration with finite products where $\dot{\times}$ preserves cocartesian arrows and $\mathbb{T} = (T, S, \mu_T, \mu_S, \eta_T, \eta_S, \mathsf{st}_T, \mathsf{st}_S)$ be a strong monad on $p$. Then, $(\widehat{S}, \widehat{\mu_S}, \widehat{\eta_S}, \widehat{\mathsf{st}_S})$ is a differential extension of $(T, \mu_T, \eta_T, \mathsf{st}_T)$.*

In other words, this result provides us with a differential extension of a monad $(T, \mu_T, \eta_T)$ starting from a usual extension $(S, \mu_S, \eta_S)$ along the fibrations $p$. Therefore, to build a differential extension of $(T, \mu_T, \eta_T)$ we can use existing techniques to lift it, obtaining a monad on the fibration $p$ and then apply our construction.

We conclude this section by instantiating this technique to a special class of bifibrations, notably, bifibrations of weak subobjects as defined in Example 3.2. The resulting construction applies to (strong) monads on categories with weak pullbacks and finite products, and provides a differential version of the Barr extension, which we dub a *differential Barr extension*.

First of all, we show that the construction of the bifibration of weak subobjects extends to a 2-functor. Let us denote by $\mathbf{Cat}_{cwp}$ the 2-category of categories with weak pullbacks and finite products, functors, and natural transformations. Given a functor $F\colon \mathcal{C} \to \mathcal{D}$ in $\mathbf{Cat}_{cwp}$, define $\overline{F}\colon \mathsf{ws}(\mathcal{C}) \to \mathsf{ws}(\mathcal{D})$ as $\overline{F}(X, [\alpha]) = (FX, [F\alpha])$ and $\overline{F}f = Ff$. It is easy to check that this is indeed a functor. Moreover, note that if $f\colon (X, [\alpha]) \to (Y, [\beta])$ is cocartesian in $\Psi_{\mathcal{C}}$, that is, $[\beta] = [f\alpha]$, then $\overline{F}f\colon (FX, [F\alpha]) \to (FY, [F\beta])$ is cocartesian in $\Psi_{\mathcal{D}}$, as $[F\beta] = [Ff \circ F\alpha]$. Consider now a natural transformation $\phi : F \xrightarrow{\cdot} G$ in $\mathbf{Cat}_{cwp}$ and define $\overline{\phi}_{(X,[\alpha])} : \overline{F} \xrightarrow{\cdot} \overline{G}$ as $\overline{\phi}_{(X,[\alpha])} = \phi_X$. This is well-defined because, if $\alpha\colon A \to X$, then we have $\phi_X \circ F\alpha = G\alpha \circ \phi_A$, by naturality of $\phi$. We define $\Psi_F = (F, \overline{F})$ and $\Psi_\phi = (\phi, \overline{\phi})$.

▶ **Proposition 4.14.** *$\Psi\colon \mathbf{Cat}_{cwp} \to \mathbf{biFib}_c$ is a strict 2-functor.*

Therefore, composing $\Psi$ and DR we get a lax functor from $\mathbf{Cat}_{cwp}$ to $\mathbf{Fib}$, this way extending Theorem 4.13.

▶ **Theorem 4.15.** *Let $\mathbb{T} = (T, \mu, \eta, \mathsf{st})$ be a strong monad on a category $\mathcal{C}$ with weak pullbacks and finite products. Then, $(\widehat{\overline{T}}, \widehat{\overline{\mu}}, \widehat{\overline{\eta}}, \widehat{\overline{\mathsf{st}}})$ is a differential extension of $\mathbb{T}$ along $\Psi_{\mathcal{C}}$.*

▶ **Example 4.16.** Let $(T, \mu, \eta)$ a monad on $\mathcal{S}et$. The *coupling-based lifting* of $(T, \mu, \eta)$ to $\mathsf{dr}(\mathsf{Sub}_{\mathcal{S}et})$ [25] maps an object $(X, V, R)$ to $(TX, T(X \times V), \tau(R))$ where $(x, v, y) \in \tau(R)$ iff there exists $\varphi \in TR$ such that $T\pi_1(T\iota_R(\varphi)) = x$, $T\langle \pi_1, \pi_2\rangle(T\iota_R(\varphi)) = v$, and $T\pi_3(T\iota_R(\varphi)) = y$. Here, $\iota_R\colon R \to X \times V \times X$ denotes the inclusion function and the element $\varphi$ is called a (ternary) coupling, borrowing the terminology from optimal transport [96]. Basically, this lifting states that monadic elements $x$ and $y$ are related with monadic distance $v$ iff we can find a coupling $\varphi$ that projected along the first and third component gives $x$ and $y$, respectively, and projected along the first two components gives the monadic distance $v$. Relying on the equivalence between $\mathsf{Sub}_{\mathcal{S}et}$ and $\Psi_{\mathcal{S}et}$, we get that the above lifting of $(T, \mu, \eta)$ is an instance of the differential Barr extension $\widehat{\overline{T}}\colon \mathsf{dr}(\mathsf{ws}(\mathcal{S}et)) \to \mathsf{dr}(\mathsf{ws}(\mathcal{S}et))$. Indeed, for an object $(X, V, [\alpha])$ in $\mathsf{dr}(\Psi_{\mathcal{S}et})$ we can choose a canonical representative $\iota_R\colon R \to X \times V \times X$ such that $[\alpha] = [\iota_R]$, where $R$ is the image of $\alpha$ and $\iota_R$ is the inclusion function.

## 5 Conclusion and Future Work

We have shown how fibrations can be used to give a uniform account to operational logical relations for higher-order languages with generic effects and a rather liberal operational semantics. Our framework encompasses both traditional, set-based logical relations and logical relations on non-cartesian-closed categories – such as the one of measurable spaces – as well as the recently introduced differential logical relations. In particular, our analysis sheds

a new light on the mathematical foundation of both pure and effectful differential logical relations. Further examples of logical relations that can be described in our framework include classic Kripke logical relations [74] (take fibrations of poset-indexed relations) as well as logical relations for information flow (the latter could be approached both as suitable Kripke logical relations or by considering a shallow semantics on the category of classified sets [63]). Additionally, since differential logical relations can be used to reason about nontrivial notions such as program sensitivity and cost analysis [25], our framework can be used for reasoning about the same notions too.

Even if general, the vehicle calculus of this work lacks some important programming language features, such as recursive types and polymorphism. Our framework being operational, the authors suspect that the addition of polymorphism should not be problematic, whereas the addition of full recursion may require to come up with abstract notions of step-indexed logical relations [5, 32]. In general, the proposed framework looks easily extensible and adaptable. To reason about a given calculus, one should pick a fibration with a logical structure supporting the kind of analysis one is interested in and whose base category has enough structure to model its interactive behaviour. For instance, in this work we just need products and a monad on the base category but, e.g., to model sum types, we would need also coproducts to describe case analysis. On the logical side, we have just considered standard intuitionistic connectives but, e.g., to support more quantitative analysis, one may need to use fibrations supporting linear connectives and modalities [85, 72, 67, 19, 18].

Besides the extension of our framework to richer languages and features, an interesting direction for future work is to formally relate our results with general theories of denotational logical relations. Particularly relevant for that seems the work by Katsumata [58] who observes that the closed structure of base categories is not necessarily essential. Another interesting direction for future work is the development of fibrational theories of coinductive reasoning for higher-order languages. Fibrational accounts of coinductive techniques have been given in the general setting of coalgebras [15, 14], whereas general accounts of coinductive techniques for higher-order languages have been obtained in terms of relational reasoning [28, 27, 37, 21, 20, 29, 24, 38, 26]. It would be interesting to see whether these two lines of research could be joined in our fibrational framework. That may also be a promising path to the development of conductive differential reasoning.

## References


1. Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proc. ACM Program. Lang.*, 4(ICFP):90:1–90:28, 2020. doi:10.1145/3408972.
2. S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.
3. Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proc. of ESOP 2006*, pages 69–83, 2006. doi:10.1007/11693024_6.
4. Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.
5. A.W. Appel and D.A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi:10.1145/504709.504712.
6. Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961.
7. H.P. Barendregt. *The lambda calculus: its syntax and semantics.* Studies in logic and the foundations of mathematics. North-Holland, 1984.
8. M. Barr. Relational algebras. *Lect. Notes Math.*, 137:39–55, 1970.

**9** Jean Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77. Springer, 1967. doi:10.1007/BFb0074299.

**10** Nick Benton, Martin Hofmann, and Vivek Nigam. Abstract effects and proof-relevant logical relations. In *Proc. of POPL 2014*, pages 619–632, 2014.

**11** 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.

**12** Lars Birkedal, Guilhem Jaber, Filip Sieczkowski, and Jacob Thamsborg. A kripke logical relation for effect-based program transformations. *Inf. Comput.*, 249:160–189, 2016.

**13** A. Bizjak and L. Birkedal. Step-indexed logical relations for probability. In *Proc. of FOSSACS 2015*, pages 279–294, 2015. doi:10.1007/978-3-662-46678-0_18.

**14** Filippo Bonchi, Barbara König, and Daniela Petrisan. Up-to techniques for behavioural metrics via fibrations. In *Proc. of CONCUR 2018*, pages 17:1–17:17, 2018.

**15** Filippo Bonchi, Daniela Petrisan, Damien Pous, and Jurriaan Rot. Coinduction up-to in a fibrational setting. In *Proc. of CSL-LICS '14*, pages 20:1–20:9, 2014.

**16** Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 33–46, 2016.

**17** Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electr. Notes Theor. Comput. Sci.*, 172:259–299, 2007.

**18** Francesco Dagnino and Fabio Pasquali. Logical foundations of quantitative equality. In *Proceedings of the 37th ACM/IEEE Symposium on Logic in Computer Science, LICS 2022*. ACM, 2022. to appear. doi:10.1145/3531130.3533337.

**19** Francesco Dagnino and Giuseppe Rosolini. Doctrines, modalities and comonads. *Mathematical Structures in Computer Science*, pages 1–30, 2021. doi:10.1017/S0960129521000207.

**20** Ugo Dal Lago and Francesco Gavazzo. Effectful normal form bisimulation. In *Proc. of ESOP 2019*, pages 263–292, 2019. doi:10.1007/978-3-030-17184-1_10.

**21** Ugo Dal Lago and Francesco Gavazzo. On bisimilarity in lambda calculi with continuous probabilistic choice. In *Proc. of MFPS 2019*, pages 121–141, 2019. doi:10.1016/j.entcs.2019.09.007.

**22** Ugo Dal Lago and Francesco Gavazzo. Differential logical relations part II: increments and derivatives. In Gennaro Cordasco, Luisa Gargano, and Adele A. Rescigno, editors, *Proc. of ICTCS 2020*, volume 2756 of *CEUR Workshop Proceedings*, pages 101–114, 2020.

**23** Ugo Dal Lago and Francesco Gavazzo. Differential logical relations, part II increments and derivatives. *Theor. Comput. Sci.*, 895:34–47, 2021.

**24** Ugo Dal Lago and Francesco Gavazzo. Resource transition systems and full abstraction for linear higher-order effectful programs. In *Proc. of FSCD 2021*, volume 195 of *LIPIcs*, pages 23:1–23:19, 2021. doi:10.4230/LIPIcs.FSCD.2021.23.

**25** Ugo Dal Lago and Francesco Gavazzo. Effectful program distancing. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

**26** Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022.

**27** 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. doi:10.1109/LICS.2017.8005117.

**28** Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for call-by-name lambda calculi. In *Proc. of ICTCS 2017*, pages 87–98, 2017.

**29** Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for call-by-name lambda calculi. *Theor. Comput. Sci.*, 813:234–247, 2020. doi:10.1016/j.tcs.2019.12.025.

**30** Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. Differential logical relations, part I: the simply-typed case. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and

Stefano Leonardi, editors, *Proc. of ICALP 2019*, volume 132 of *LIPIcs*, pages 111:1–111:14, 2019.

31 Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. Differential logical relations, part I: the simply-typed case. In *Proc. of ICALP 2019*, pages 111:1–111:14, 2019. doi:10.4230/LIPIcs.ICALP.2019.111.

32 Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

33 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.

34 Brian P. Dunphy and Uday S. Reddy. Parametric limits. In *Proc. of LICS 2004*, pages 242–251. IEEE Computer Society, 2004.

35 Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *Proc. of POPL 2017*, 2:1–28, 2017.

36 Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. *PACMPL*, 2(POPL):59:1–59:28, 2018.

37 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In *Proc. of LICS 2018*, pages 452–461, 2018. doi:10.1145/3209108.3209149.

38 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/.

39 Neil Ghani, Patricia Johann, Fredrik Nordvall Forsberg, Federico Orsanigo, and Tim Revell. Bifibrational functorial semantics of parametric polymorphism. In Dan R. Ghica, editor, *Proc. of MFPS 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 165–181. Elsevier, 2015.

40 Neil Ghani, Patricia Johann, and Clément Fumex. Fibrational induction rules for initial algebras. In Anuj Dawar and Helmut Veith, editors, *Proc. of CSL 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2010.

41 Neil Ghani, Patricia Johann, and Clément Fumex. Generic fibrational induction. *Log. Methods Comput. Sci.*, 8(2), 2012.

42 Neil Ghani, Patricia Johann, and Clément Fumex. Indexed induction and coinduction, fibrationally. *Log. Methods Comput. Sci.*, 9(3), 2013.

43 J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

44 Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, pages 68–85. Springer Berlin Heidelberg, 1982.

45 A.D. Gordon. A tutorial on co-induction and functional programming. In *Workshops in Computing*, pages 78–95. Springer London, September 1994. doi:10.1007/978-1-4471-3573-9_6.

46 Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. Logical relations for monadic types. *Math. Struc. Comput. Sci.*, 18(6):1169–1217, 2008. doi:10.1017/S0960129508007172.

47 Marco Grandis. Weak subobjects and the epi-monic completion of a category. *Journal of Pure and Applied Algebra*, 154(1-3):193–212, 2000.

48 Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016.

49 Claudio Hermida. *Fibrations, logical predicates and indeterminates*. PhD thesis, University of Edinburgh, UK, 1993.

50 Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

**51** Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity - A reynolds programme for category theory and programming languages. *Electronic Notes on Theoretical Computer Science*, 303:149–180, 2014. doi:10.1016/j.entcs.2014.02.008.

**52** Martin Hofmann. Logical relations and nondeterminism. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, pages 62–74, 2015.

**53** Pieter Hofstra. The dialectica monad and its cousins. In *Models, Logics, and Higher-dimensional Categories: A Tribute to the Work of Mihaly Makkai*, number 53 in CRM proceedings & lecture notes, pages 107–139, 2011.

**54** Jesse Hughes and Bart Jacobs. Factorization systems and fibrations: Toward a fibred birkhoff variety theorem. In Richard Blute and Peter Selinger, editors, *CTCS 2002*, volume 69 of *Electronic Notes in Theoretical Computer Science*, pages 156–182. Elsevier, 2002.

**55** Bart P. F. Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in logic and the foundations of mathematics*. North-Holland, 2001. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780444508539.

**56** 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. doi:10.1109/LICS.2010.29.

**57** Ohad Kammar and Dylan McDermott. Factorisation systems for logical relations and monadic lifting in type-and-effect system semantics. In Sam Staton, editor, *Proc. of MFPS 2018*, volume 341 of *Electronic Notes in Theoretical Computer Science*, pages 239–260. Elsevier, 2018.

**58** Shin-ya Katsumata. *A generalisation of pre-logical predicates and its applications*. PhD thesis, University of Edinburgh, UK, 2005.

**59** Shin-ya Katsumata. A semantic formulation of tt-lifting and logical predicates for computational metalanguage. In C.-H. Luke Ong, editor, *Proc. of CSL 2005*, volume 3634 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2005.

**60** Shin-ya Katsumata. Relating computational effects by $\top\top$-lifting. *Inf. Comput.*, 222:228–246, 2013.

**61** Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of POPL 2014*, pages 633–646, 2014.

**62** Shin-ya Katsumata, Tetsuya Sato, and Tarmo Uustalu. Codensity lifting of monads and its dual. *Logical Methods in Computer Science*, 14(4), 2018.

**63** G. A. Kavvos. Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.*, 3(POPL):20:1–20:29, 2019.

**64** Gregory M. Kelly. Basic concepts of enriched category theory. *Reprints in Theory and Applications of Categories*, (10):1–136, 2005.

**65** Daan Leijen. Implementing algebraic effects in c. In Bor-Yuh Evan Chang, editor, *Programming Languages and Systems*, pages 339–363, Cham, 2017. Springer International Publishing.

**66** P.B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

**67** Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017*, volume 84 of *LIPIcs*, pages 25:1–25:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.25.

**68** QingMing Ma and John C. Reynolds. Types, abstractions, and parametric polymorphism, part 2. In *Proc. of MFPS 1991*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 1991.

**69** Maria Emilia Maietti, Fabio Pasquali, and Giuseppe Rosolini. Quasi-toposes as elementary quotient completions, 2021. arXiv:arXiv:2111.15299.

**70** Maria Emilia Maietti and Giuseppe Rosolini. Elementary quotient completion. *Theory and Application of Categories*, 27(17):445–463, 2013.

71 Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Ssemantics*. Texts and Monographs in Computer Science. Springer, 1986.
72 Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM Symposium on Principles of Programming Languages, POPL 2015*, pages 3–16. ACM, 2015. doi:10.1145/2676726.2676970.
73 Paul-André Melliès and Noam Zeilberger. A bifibrational reconstruction of lawvere's presheaf hyperdoctrine. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*, pages 555–564. ACM, 2016. doi:10.1145/2933575.2934525.
74 John C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.
75 John C. Mitchell and Andre Scedrov. Notes on sconing and relators. In *Proc. of CSL '92*, volume 702 of *Lecture Notes in Computer Science*, pages 352–378. Springer, 1992.
76 Peter W. O'Hearn and Robert D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.
77 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
78 Paolo Pistone. On generalized metric spaces for the simply typed lambda-calculus. In *Proc. of LICS 2021*, pages 1–14, 2021.
79 Gordon Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4, School of A.I., University of Edinburgh, 1973.
80 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *Proc. of FOSSACS 2001*, pages 1–24, 2001. doi:10.1007/3-540-45315-6_1.
81 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.
82 J. Reed and B.C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. of ICFP 2010*, pages 157–168, 2010. doi:10.1145/1863543.1863568.
83 J.C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
84 Edmund P. Robinson and Giuseppe Rosolini. Reflexive graphs and parametric polymorphism. In *Proc. of LICS '94*, pages 364–371. IEEE Computer Society, 1994.
85 Michael A. Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, (18):650–738, 2008.
86 Kristina Sojakova and Patricia Johann. A general framework for relational parametricity. In Anuj Dawar and Erich Grädel, editors, *Proce. of LICS 2018,*, pages 869–878. ACM, 2018.
87 Richard Statman. Logical relations and the typed lambda-calculus. *Inf. Control.*, 65(2/3):85–97, 1985.
88 Sam Staton. Commutative semantics for probabilistic programming. In *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*, pages 855–879, 2017.
89 Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. 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 525–534, 2016.
90 Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972. doi:10.1016/0022-4049(72)90019-9.
91 Ross Street. Two constructions on lax functors. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 13(3):217–264, 1972.
92 Thomas Streicher. Fibered categories a la jean benabou, 2018. arXiv:arXiv:1801.02927.
93 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.

**94** Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In Roberto Giacobazzi and Radhia Cousot, editors, *Proc. of POPL '13*, pages 343–356. ACM, 2013.

**95** Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL):36:1–36:29, 2019.

**96** C. Villani. *Optimal Transport: Old and New*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008.

**97** Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *PACMPL*, 2(ICFP):87:1–87:30, 2018.

# On Quantitative Algebraic Higher-Order Theories

**Ugo Dal Lago** ✉
Department of Computer Science and Engineering, University of Bologna, Italy

**Furio Honsell** ✉
Department of Mathematical Sciences, Informatics and Physics, University of Udine, Italy

**Marina Lenisa** ✉
Department of Mathematical Sciences, Informatics and Physics, University of Udine, Italy

**Paolo Pistone** ✉
Department of Computer Science and Engineering, University of Bologna, Italy

**Abstract**

We explore the possibility of extending Mardare *et al.*'s quantitative algebras to the structures which naturally emerge from Combinatory Logic and the $\lambda$-calculus. First of all, we show that the framework is indeed applicable to those structures, and give soundness and completeness results. Then, we prove some negative results clearly delineating to which extent categories of metric spaces can be models of such theories. We conclude by giving several examples of non-trivial higher-order quantitative algebras.

**2012 ACM Subject Classification** Theory of computation → Program semantics; Theory of computation → Type theory

**Keywords and phrases** Quantitative Algebras, Lambda Calculus, Combinatory Logic, Metric Spaces

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.4

**Related Version** *Full Version*: https://arxiv.org/abs/2204.13654

**Funding** Ugo Dal Lago's and Paolo Pistone's work was funded by ERC CoG 818616.

## 1 Introduction

One way of seeing program semantics is as the science of program equivalence. Each way of giving semantics to programs implicitly identifies which programs are equivalent. Similarly, a notion of program equivalence can be seen as a way of attributing meaning to programs (namely, the equivalence class to which the program belongs). This point of view makes semantics a powerful source of ideas and techniques for program transformation and program verification, with the remarkable advantage that such techniques can be defined in a compositional and modular way.

However, there are circumstances in which equivalences between programs, being purely dychotomous, are just not informative enough: two programs are either equivalent or not, period. No further quantitative or causal information can be extracted from two programs which are *slightly* different, although not equivalent. Furthermore, as program equivalences are usually congruences, and therefore preserved by any context, programs that only differ in *peculiar* circumstances are also just non-equivalent. For these reasons, methods alternative to program equivalence have to be looked for in all (very common) situations involving transformations that replace a program by one which is only *approximately* equivalent [31], or when the specifications are either not precise or not to be met precisely (e.g. in modern cryptography [27], in which most security properties hold in an approximate sense, namely modulo a negligible probability).

© Ugo Dal Lago, Furio Honsell, Marina Lenisa, and Paolo Pistone;
licensed under Creative Commons License CC-BY 4.0

The considerations above led the scientific community to question the possibility of broadening the scope of program semantics from a science of equivalences to a science of *distances* between programs. By the way, the possibility of interpreting programs in domains having a metric structure has been known since the 1990s [19, 18]. Recently, Mardare, Panangaden, and Plotkin have introduced a notion of quantitative algebra [29] that generalizes usual equational reasoning to a setting in which the compared entities can be at a certain distance. In this way, various notions of quantitative algebra have been shown to be captured through a formal system, *à la* Birkhoff [8].

Still, when the programs at hand are higher-order functional programs, the construction of a metric semantics faces several obstacles. First, it is well-known that the category Met of metric spaces and non-expansive maps, providing the standard setting of the approaches just recalled, is *not* a model of the simply typed $\lambda$-calculus (more precisely, it is not cartesian closed). Furthermore, finding relevant *sub*-categories of Met enjoying enough structure to model higher-order programs can lead to trivial (i.e. discrete) models, and several (mostly negative) results have remained so far in the *folklore* (with a few notable exceptions, e.g. [21]).

In this paper we bite the tail of the dragon: we apply the framework of quantitative equational theories and algebras from Mardare *et al.* to the cases of combinatory logic and the $\lambda$-calculus, and we try to highlight features and obstacles in the construction of higher-order quantitative algebras, at the same time showing the existence of several interesting models.

There are various reasons for exploring combinatory algebras, *i.e.* applicative structures where the $\xi$-rule fails. The first is that these structures naturally arise in various contexts, most notably in Game Semantics and in particular in the Geometry of Interaction [23], as axiomatized by Abramsky *et al.* [1]. The $\xi$-rule can then be enforced only by introducing a rather complex notion of equivalence relation, whose fine structure is usually rather awkward to grasp. The second reason is that combinatory algebras, being indeed algebras, might appear at first sight to be amenable straightforwardly in the first order framework of quantitative algebras of Mardare *et al.* We show that this is illusory, because the impact of the basic assumption that constructors are non-expansive, *i.e.* the Axiom NExp (see Section 4) is very strong, even in a context which could appear to be algebraically well-behaved. Finally, even if it is convenient to assume the $\xi$-rule, in reasoning about higher-order programming languages, showing that it holds in implementations is not at all immediate and, when side-effects are present, it needs to be carefully phrased.

The contributions of this paper are threefold:

- Following the framework defined by Mardare *et al.*, we introduce quantitative generalizations of the standard notions of *weak $\lambda$-theories* and *$\lambda$-theories* [6], and of their algebras. This is in Section 3, Section 4, and Section 5, respectively.
- We study properties and examples of algebras for such theories, as suitable sub-categories of Met. Notably, we highlight the relevance of *ultra*-metric and *injective* metric spaces in the construction of non-trivial (i.e. non discrete) algebras. Some examples are discussed through Section 2 and Section 5, further properties and examples are in Section 6.
- Finally, we discuss algebras obtained by relaxing the conditions from Mardare *et al.*: either by replacing metrics by *partial* metrics [9, 34], *i.e.* generalized metrics in which self-distances $d(x, x)$ need not be zero, or by relaxing the non-expansiveness condition and introducing a class of *approximate* quantitative algebras. This is in Section 7 and Section 8.

## 2 Preliminaries on Metric Spaces

In this section we discuss a few properties of metric spaces and their associated categories, which provide the general setting for quantitative algebras in the sense of Mardare *et al.* In particular, we recall the definition of ultra-metric spaces, as well as *partial ultra-metric spaces* [9, 34]. The latter is a class of generalized metric spaces in which self-distances $a(x,x)$ are not required to be 0 but only smaller than any distance of the form $a(x,y)$.

▶ **Definition 1.** *A pair $(X,a)$ formed by a set $X$ and a function $a : X \times X \to \mathbb{R}^{\infty}_{\geq 0}$ is called: (i) a* pre-metric space *if it satisfies, for all $x,y \in X$, $a(x,x) = 0$* (refl) *and $a(x,y) = a(y,x)$* (symm)*; (ii) a* (pseudo-)metric space *if it satisfies* (refl)*,* (symm)*, and, for all $x,y,z \in X$, $a(x,y) \leq a(x,z)+a(z,y)$* (trans)*; (iii) an* ultra-metric space *if it satisfies* (refl)*,* (symm) *and, for all $x,y,z \in X$, $a(x,y) \leq \max\{a(x,z), a(z,y)\}$* (trans$^*$)*; (iv) a* partial ultra-metric space *if it satisfies* (symm)*,* (trans$^*$) *and, for all $x,y \in X$, $a(x,y) \geq a(x,x), a(y,y)$* (refl$^*$)*.*

Since all metrics we consider are "pseudo", from now on we will omit this prefix. Observe that an ultra-metric space is also a metric space. Moreover, a partial ultra-metric space $(X,a)$ also yields an ultra-metric space $(X,a^*)$, with $a^*(x,y) = 0$ if $x = y$ and $a^*(x,y) = a(x,y)$ otherwise. Usually, partial metric spaces are defined using a stronger version of the triangular law, given by $a(x,y) \leq a(x,z) + a(z,y) - a(z,z)$. However, for partial *ultra*-metrics this condition is equivalent to (trans$^*$) (see e.g. [34]).

The natural morphisms to consider between metric (ultra-metric, partial ultra-metric) spaces $(X,a)$ and $(Y,b)$, hoping to get a continuous currification, are the *non-expansive* functions, i.e. those functions $f : X \to Y$ such that for all $x,y \in X$, $b(f(x),f(y)) \leq a(x,y)$. We let $\mathsf{Met}$ (resp. $\mathsf{UMet}$, $\mathsf{PUMet}$) indicate the category of metric spaces (resp. ultra-metric spaces, partial ultra-metric spaces) and non-expansive maps. All categories $\mathsf{Met}, \mathsf{UMet}$ and $\mathsf{PUMet}$ are cartesian, the product of $(X,a)$ and $(Y,b)$ being given by $(X \times Y, \max\{a,b\})$. In $\mathsf{UMet}$ and $\mathsf{PUMet}$ the cartesian functors $\{-\} \times X$ have right-adjoints given, respectively, by $(\mathsf{UMet}(X,\{-\}), \Phi_{a,\{-\}})$ and $(\mathsf{PUMet}(X,\{-\}), \Phi_{a,\{-\}})$, where for all metric space $(Y,b)$, $\Phi_{a,b}(f,g) = \sup\{b(f(x),g(x)) \mid x \in X\}$. For this reason, both categories are cartesian closed.

By contrast, $\mathsf{Met}$ is *not* cartesian closed. Indeed, the functor $(\mathsf{Met}(X,\{-\}), \Phi_{a,\{-\}})$ is right-adjoint in $\mathsf{Met}$ (and thus also in $\mathsf{UMet}$) to the functor $(X \times \{-\}, a + \{-\})$, but for all metric spaces $(Y,b)$, $(X \times Y, a+b)$ is isomorphic to the cartesian product $(X \times Y, \max\{a,b\})$ only when $X$ and $Y$ are ultra-metrics. Instead, the exponential of $(X,a)$ and $(Y,b)$ in $\mathsf{Met}$, if it exists, is necessarily of the form $(\mathsf{Met}(X,Y), \Xi_{a,b})$ (as shown in the long version), where

$$\Xi_{a,b}(f,g) = \inf\{\delta \mid \forall x,y \in X \ \ \max\{\delta, a(x,y)\} \geq b(f(x),g(y))\}$$

We use the Greek letter $\Xi$, since, as we'll see, this metric is tightly related to the interpretation of the $\xi$-rule of the $\lambda$-calculus. Notice that in general $\Xi_{a,b}$ is only a pre-metric. Indeed, the category of pre-metric spaces and non-expansive functions is cartesian closed, while the exponential of $(X,a)$ and $(Y,b)$ exists in $\mathsf{Met}$ precisely when $\Xi_{a,b}$ further satisfies (trans).

We will exploit the following useful characterization of exponentiable objects in $\mathsf{Met}$ (we recall that an object $A$ in a cartesian category $\mathbb{C}$ is exponentiable when, for all object $B$, the exponential of $B$ and $A$ exists in $\mathbb{C}$, so $\mathbb{C}$ is cartesian closed iff all its objects are exponentiable):

▶ **Theorem 2** ([13])**.** *A metric space $(X,a)$ is exponentiable in* $\mathsf{Met}$ *iff for all $x_0,x_2 \in X$ and $\alpha,\beta \in \mathbb{R}^{\infty}_{\geq 0}$ such that $a(x_0,x_2) = \alpha+\beta$, the condition below holds:*

$$\forall \epsilon > 0 \ \exists x_1 \in X \ \text{s.t.} \ a(x_0,x_1) < \alpha+\epsilon \ \text{and} \ a(x_1,x_2) < \beta+\epsilon \qquad (*)$$

Condition $(*)$ intuitively requires $X$ to have "enough points". For example, the set $\mathbb{N}$, as a subspace of $\mathbb{R}$, is not exponentiable in $\mathsf{Met}$ (take $x_0 = 0, x_1 = 1$ and $\alpha = \beta = 1/2$: a point between 0 and 1 is "missing"). Instead, condition $(*)$ always holds when $(X, a)$ is *injective* (see [22, 13]): for any collection of points $\{x_i\}_{i\in I}$ in $X$ and positive reals $\{r_i\}_{i\in I}$ such that $a(x_i, x_j) \leq r_i + r_j$, there is a point lying in the intersection of all balls $B(x_i, r_i)$. This implies that the sub-category $\mathsf{InjMet}$ of $\mathsf{Met}$ formed by injective metric spaces is cartesian closed. Since the Euclidean metric is injective, there is a cartesian closed sub-category of $\mathsf{Met}$ formed by "simple types" over closed real intervals, that we'll use as working example.

▶ **Example 3.** Let IntST be the set of *simple types over the intervals*, defined by $[a, b] \in \mathrm{IntST}$, for all intervals $[a, b]$ (with $a, b \in \mathbb{R}^{\infty}_{\geq 0}$ and $a \leq b$) and $i, j \in \mathrm{IntST} \Rightarrow (i \times j), (i \to j) \in \mathrm{IntST}$. For any $i \in \mathrm{IntST}$, the metric spaces $(\mathcal{I}_i, d^{\mathcal{I}}_i)$ are defined by $\mathcal{I}_{[a,b]} := [a, b]$, $\mathcal{I}_{i\times j} := \mathcal{I}_i \times \mathcal{I}_j$, $\mathcal{I}_{i\to j} := \mathsf{Met}(\mathcal{I}_i, \mathcal{I}_j)$, $d^{\mathcal{I}}_{[a,b]}(x, y) := |x - y|$, $d^{\mathcal{I}}_{i\times j} := \max\{d^{\mathcal{I}}_i, d^{\mathcal{I}}_j\}$ and $d^{\mathcal{I}}_{i\to j} := \Xi_{d^{\mathcal{I}}_i, d^{\mathcal{I}}_j}$.

The "analytic knife" provided by metrics is rather blunt when dealing with isometries in $\mathbb{R}$, because these are isolated points in $\Xi_{a,b}$. Examples of isometries are the identity and functions which have a right or left inverse. We have:

▶ **Proposition 4.** *Let $f : \mathbb{R} \to \mathbb{R}$ be an isometry in $\mathbb{R}$. Then $f$ is isolated in the metric $\Xi_{a,b}$. Moreover the identity is isolated in all injective spaces.*

## 3 Many-Sorted Quantitative Theories and Algebras

In this section we introduce quantitative theories and algebras in the sense of [29]. In order to cover both the typed and the untyped case, we consider *many-sorted* theories and algebras, hence combining the quantitative (but one-sorted) approach from [29] with the qualitative (but many-sorted) approach from [24].

**Notation.** For any set $I$, an *$I$-sorted set* $A$ is an $I$-indexed family of sets $A = (A_i)_{i\in I}$ (i.e. an object of $\mathsf{Set}^I$), and an *$I$-sorted function* $f : A \to B$ between $I$-sorted sets is an $I$-indexed family of functions $f = (f_i : A_i \to B_i)_{i\in I}$ (i.e. a morphism in $\mathsf{Set}^I(A, B)$). For a set $I$, we denote by $I^*$ the set of all finite lists of elements of $I$, we let $w$ range over $I^*$ and use $*$ for concatenation. For $A$ an $I$-sorted set and $w = i_1 \dots i_k \in I^*$, we let $A_w := \prod_{j=1}^{k} A_{i_j}$. Var denotes a distinguished $I$-sorted containing, such that for all $i \in I$, $\mathrm{Var}_i$ is a countably infinite set of variables. For any $I$-sorted set $A$ and function $f : \mathrm{Var} \to A$, and pairwise disjoint variables $x_1, \dots, x_n$, with $x_j \in \mathrm{Var}_{i_j}$ and $a_1, \dots, a_n$ with $a_j \in A_{i_j}$, we let $f_{\vec{x},\vec{a}} : \mathrm{Var} \to A$ indicate the $I$-sorted function mapping $x_j$ to $a_j$ and behaving as $f$ on all other variables.

▶ **Definition 5** (Many-Sorted Signature). *An* $I$-sorted signature $\Sigma$ *is an $I^* \times I$-sorted set $\{\Sigma_{w,i} \mid w \in I^*, i \in I\}$ (i.e. an object of $\mathsf{Set}^{I^*\times I}$).*

The objects $\sigma \in \Sigma_{w,i}$ will be called *symbols* of the signature.

▶ **Definition 6** ($\Sigma$-Algebra). *A* $\Sigma$-algebra *is a pair $(A, \Omega^A)$ where $A$ is a $I$-sorted family and $\Omega^A$ associates each symbol $\sigma \in \Sigma_{w,i}$ with a function $\sigma_A : A_w \to A_i$, where $A_w = A_{i_1} \times \dots \times A_{i_k}$, for $w = i_1 \dots i_k$. For any object $A$ of $\mathsf{Set}^I$, the* free $\Sigma$-algebra over $A$*, noted $\mathbb{F}_\Sigma(A)$, is the $I$-sorted set defined by the following conditions: (i) for all $x \in A_i$, $x \in \mathbb{F}_\Sigma(A)_i$; (ii) for all $\sigma \in \Sigma_{w,i}$ and $v_1 \in \mathbb{F}_\Sigma(A)_{w(1)}, \dots, v_k \in \mathbb{F}_\Sigma(A)_{w(k)}$, then $\sigma_{\mathbb{F}_\Sigma(A)}(v_1, \dots, v_k) := \sigma(v_1, \dots, v_k) \in \mathbb{F}_\Sigma(A)_i$.*

Intuitively, $\mathbb{F}_\Sigma(A)_i$ is the set of "terms of sort $i$ with parameters in $A$". Free algebras enjoy the following universal property:

▶ **Proposition 7.** *For any* $\Sigma$*-algebra* $(A, \Omega^A)$ *and map* $f \in \mathsf{Set}^I(B, A)$ *there exists a unique* $\Sigma$*-homomorphism* $f^\sharp : \mathbb{F}_\Sigma(B) \to A$ *extending* $f$*, that is, such that* $f = f^\sharp \circ \eta_B$*, where* $\eta_B : B \to \mathbb{F}_\Sigma(B)$ *is the inclusion map.*

Given a function $f \in \mathsf{Set}^I(B, A)$, if $t \in \mathbb{F}_\Sigma(B)_i$ is some term of sort $i$ with parameters $b_1, \dots, b_n$ in $B$, $f^\sharp t \in \mathbb{F}_\Sigma(A)_i$ is the result of "substituting" each parameter $b_i$ in $t$ with $f(b_i)$.

Let us now introduce the equational language of quantitative theories.

▶ **Definition 8.** *Let* $\Sigma$ *be an* $I$*-sorted signature.*

**(i)** *A* quantitative $\Sigma$-equation *over* $\mathbb{F}_\Sigma(\mathrm{Var})$ *is an expression of the form* $t \overset{i}{\simeq}_\epsilon s$*, where* $i \in I$*,* $t, s \in \mathbb{F}_\Sigma(\mathrm{Var})_i$ *and* $\epsilon \in \mathbb{Q}_{\geq 0}$*.*

**(ii)** *For all* $\epsilon \in \mathbb{Q}_{\geq 0}$*, let* $\mathcal{V}(\mathrm{Var})$ *be the set of* indexed $\Sigma$-equations *of the form* $x \overset{i}{\simeq}_\epsilon y$*, for some* $i \in I$ *and* $x, y \in \mathrm{Var}_i$*, and* $\mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))$ *be the set of indexed* $\Sigma$*-equations of the form* $t \overset{i}{\simeq}_\epsilon s$*, where* $i \in I$ *and* $t, s \in (\mathbb{F}_\Sigma(\mathrm{Var}))_i$*.*

▶ **Definition 9.** *A* consequence relation *on the free* $\Sigma$*-algebra* $\mathbb{F}_\Sigma(\mathrm{Var})$ *is a relation* $\vdash \subseteq \wp(\mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))) \times \mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))$ *closed under all instances of the following rules (where* $\epsilon, \delta$ *vary over all* $\mathbb{Q}_{\geq 0}$*):*

***(Cut)*** *if* $\Gamma \vdash \phi$ *for all* $\phi \in \Gamma'$ *and* $\Gamma' \vdash \psi$*, then* $\Gamma \vdash \psi$*;*
***(Assumpt)*** *if* $\phi \in \Gamma$*, then* $\Gamma \vdash \phi$*;*
***(Refl)*** $\emptyset \vdash t \overset{i}{\simeq}_0 t$*;*
***(Symm)*** $\{t \overset{i}{\simeq}_\epsilon s\} \vdash s \overset{i}{\simeq}_\epsilon t$*;*
***(Triang)*** $\{t \overset{i}{\simeq}_\epsilon s, s \overset{i}{\simeq}_\delta u\} \vdash t \overset{i}{\simeq}_{\epsilon+\delta} u$*;*
***(Max)*** $\{t \overset{i}{\simeq}_\epsilon s\} \vdash t \overset{i}{\simeq}_{\epsilon+\delta} s$*;*
***(Arch)*** $\{t \overset{i}{\simeq}_\delta s \mid \delta > \epsilon\} \vdash t \overset{i}{\simeq}_\epsilon s$ *;*
***(NExp)*** $\{t_1 \overset{i_1}{\simeq}_\epsilon s_1, \dots, t_k \overset{i_k}{\simeq}_\epsilon s_k\} \vdash \sigma(t_1, \dots, t_k) \overset{i}{\simeq}_\epsilon \sigma(s_1, \dots, s_k)$*, for all* $\sigma \in \Sigma_{i_1 \dots i_k, i}$*;*
***(Subst)*** *if* $f : \mathrm{Var} \to \mathbb{F}_\Sigma(\mathrm{Var})$*, then* $\Gamma \vdash t \overset{i}{\simeq}_\epsilon s$ *implies* $f^\sharp\Gamma \vdash f^\sharp t \overset{i}{\simeq}_\epsilon f^\sharp s$*.*
*Notice that rule (Arch) has infinitely many assumptions.*

We let $\mathcal{E}(\mathbb{F}_\Sigma(\mathrm{Var})) = \wp_{\mathrm{fin}}(\mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))) \times \mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))$ indicate the set of *quantitative inferences on* $\mathbb{F}_\Sigma(\mathrm{Var})$ and $\mathcal{E}(\mathrm{Var}) = \wp_{\mathrm{fin}}(\mathcal{V}(\mathrm{Var})) \times \mathcal{V}(\mathbb{F}_\Sigma(\mathrm{Var}))$ indicate the set of *basic quantitative inferences.* Axioms for theories will be basic quantitative inferences.

▶ **Definition 10** (Many-Sorted Quantitative Theory). *Let* $S \subseteq \mathcal{E}(\mathrm{Var})$ *be a set of basic quantitative inferences. Let* $\vdash_S$ *be the smallest consequence relation including* $S$*. The* quantitative equational theory over $\Sigma$ generated by $S$ *is the set* $\mathcal{U}_S := (\vdash_S) \cap \mathcal{E}(\mathbb{F}_\Sigma(\mathrm{Var}))$*. The elements of* $S$ *are the* axioms *of* $\mathcal{U}_S$*.*

To the syntactic notion of quantitative theory there corresponds a semantic notion of quantitative algebra, given by a $\Sigma$-algebra endowed with suitable metrics.

▶ **Definition 11** (Many-Sorted Quantitative Algebra). *Let* $\Sigma$ *be an* $I$*-sorted signature. A* quantitative $\Sigma$-algebra *is a tuple* $\mathcal{A} = (A, \Omega^{\mathcal{A}}, d^{\mathcal{A}})$ *where* $(A, \Omega^{\mathcal{A}})$ *is a* $\Sigma$*-algebra and* $d^{\mathcal{A}}$ *is an* $I$*-sorted family of metrics* $d_i^{\mathcal{A}} : A_i \times A_i \to \mathbb{R}_{\geq 0}^{\infty}$ *such that for all* $\sigma \in \Sigma_{w,i}$*,* $\sigma_A : \mathsf{Met}(A_w, A_i)$*.*

Given a quantitative $\Sigma$-algebra, we can define a multicategory $\mathsf{Met}^{\mathcal{A}}$ whose objects are the metric spaces $(A_i, d_i^{\mathcal{A}})$, and where for all $w = i_1 \dots i_k$, $\mathsf{Met}^{\mathcal{A}}(A_{i_1}, \dots, A_{i_k}; A_i) \subseteq \mathsf{Met}(A_w, A_i)$ contains all functions $f \in \mathsf{Met}(A_w, A_i)$ such that for some term $t_f \in \mathbb{F}_\Sigma(A + \{x_1 : w(1), \dots, x_k : w(k)\})$, $f(a_1, \dots, a_k) = f^\sharp_{\vec{x}, \vec{a}}(t_f)$. For brevity, we will often abbreviate $\mathsf{Met}^{\mathcal{A}}(A_{i_1}, \dots, A_{i_k}; A_i)$ as $\mathsf{Met}^{\mathcal{A}}(A_w; A_i)$.

▶ **Definition 12.** *Let $\mathcal{A} = (A, \Omega^{\mathcal{A}}, d^{\mathcal{A}})$ be a quantitative $\Sigma$-algebra. For any $f : \mathrm{Var} \to A$, we say that $\mathcal{A}$ satisfies a quantitative equation $\phi = t \overset{i}{\simeq}_\epsilon u$* relative to $f$ *(denoted $\vDash^f_{\mathcal{A}} \phi$) when $d^{\mathcal{A}}_i(f^\sharp(t), f^\sharp(u)) \leq \epsilon$. We say that $\mathcal{A}$ satisfies a quantitative inference $\Gamma \vdash \phi$ (denoted $\Gamma \vDash_{\mathcal{A}} \phi$) if for all $f : \mathrm{Var} \to A$, if $\vDash^f_{\mathcal{A}} \psi$ holds for all $\psi \in \Gamma$, then $\vDash^f_{\mathcal{A}} \phi$ also holds.*

Notice that the interpretation of rule (Nexp) implies that functional terms need to be interpreted as non-expansive morphisms.

▶ Remark 13. All constructions from this section can be adapted to the case of *partial* ultra-metric spaces by replacing, in Definition 9, the rule (Refl) with the following rule:
**(PRefl)** $\{t \overset{i}{\simeq}_\epsilon u\} \vdash t \overset{i}{\simeq}_\epsilon t$;
and requiring in Def. 11 that the $d^{\mathcal{A}}_i$ are partial ultra-metrics and $\sigma^{\mathcal{A}} \in \mathsf{PUMet}(A_w, A_i)$.

## 4 Quantitative Weak λ-Theories and Algebras

As is well-known (see e.g. [5]), a purely algebraic approach to the λ-calculus is provided by *combinatory logic* **CL**. Hence, it is natural to start from this calculus. The equational theory of **CL** captures so-called *weak λ-theories* [5], namely λ-theories where the *ξ-rule* (discussed in more detail in Section 5) may fail. In this section we introduce quantitative weak λ-theories and we discuss their algebras, of which $\mathsf{Met}$ itself is a notable example.

▶ **Definition 14** (Applicative Signature). *Let $T$ be a set of sorts (called* types*) endowed with a binary function $\to : T \times T \to T$. An* applicative signature *$\Sigma$ is a $T$-sorted signature which includes symbols $\cdot_{i,j} \in \Sigma_{(i \to j)*i,j}$, for all $i, j \in T$.*

We will often note $\cdot_{i,j}(t, u)$ infix, *i.e.* $t \cdot_{i,j} u$, or simply as $tu$, when clear from the context. For all $w = i_1 \dots i_n \in T^*$ and $j \in T$, we let $w \to j := i_1 \to \cdots \to i_n \to j$. A notable example of applicative signature is the following:

▶ **Definition 15** (**CL**-Signature). *Let $\Sigma^{\mathbf{CL}}$ be the applicative signature which includes symbols $\mathsf{I}_i : i \to i$, $\mathsf{K}_{ij} : i \to j \to i$, $\mathsf{S}_{ijk} : (i \to j \to k) \to (i \to j) \to (i \to k)$, for all $i, j, k \in T$. The* terms of combinatory logic *are the elements of the free $\Sigma^{\mathbf{CL}}$-algebra, $\mathbb{F}_{\mathbf{CL}}(\mathrm{Var})$.*

Definition 15 above comprises both the typed and untyped case. In typed Combinatory Logic the set of types $T$ includes at least a *base type $o$*, i.e. *a type which is not in the image of $\to$ and $\to$ is injective*, while in the untyped case $T$ is a singleton set $\{\star\}$ and hence $\star \to \star = \star$. In the traditional language of "syntax and semantics", used for instance in [5], when $f : \mathrm{Var} \to A$, the function $f^\sharp$ of Proposition 7, amounts to the notion of *intepretation* of a term $t$ in the environment $f$, namely $f^\sharp(t) = [\![t]\!]_f$ .

We now introduce the natural notion of theory for a **CL**-signature:

▶ **Definition 16** (**CL**-Theory). *The quantitative equational theory over $\mathbb{F}_{\mathbf{CL}}(\mathrm{Var})$, $\mathcal{U}_{\mathbf{CL}}$ is generated by the axioms $\emptyset \vdash \mathsf{I}_i t \overset{i}{\simeq}_0 t$, $\emptyset \vdash \mathsf{K}_{ij} tu \overset{i}{\simeq}_0 t$, and $\emptyset \vdash \mathsf{S}_{ijk} tuw \overset{k}{\simeq}_0 tw(uw)$. We call* (quantitative) weak λ-theory *any theory including $\mathcal{U}_{\mathbf{CL}}$.*

▶ **Example 17.** The set IntST (cf. Example 3) is a particular instance of the set $T$. Let $\mathcal{I}(\Sigma^{\mathbf{CL}})$ be the signature obtained by enriching $\Sigma^{\mathbf{CL}}$ with 0-ary symbols $\overline{r} \in \mathcal{I}(\Sigma)_{(),[a,b]}$ for all $r \in [a, b]$, and $k$-ary symbols $\overline{f} \in \mathcal{I}(\Sigma)_{[a_1,b_1],\dots,[a_n,b_n],[a,b]}$ for all $f \in \mathsf{Met}(\prod_i [a_i, b_i], [a, b])$. Let $\mathcal{U}^{\mathcal{I}}_{\mathbf{CL}}$ be the theory obtained by extending $\mathcal{U}_{\mathbf{CL}}$ with all axioms $\emptyset \vdash \overline{f}\,\overline{r}_1 \dots \overline{r}_k \overset{[a,b]}{\simeq}_0 \overline{s}$ whenever $f(r_1, \dots, r_k) = s$ as well as all axioms $\emptyset \vdash \overline{r} \overset{[a,b]}{\simeq}_\epsilon \overline{s}$ for all rational $\epsilon \geq |r - s|$.

A well-known property of Combinatory Logic is *functional completeness*: for any term $t$ and variable $x$, one can construct a term $\Lambda_x(t)$ so that $\Lambda_x(t)$ "simulates" $\lambda$-abstraction in the sense that one can prove $\Lambda_x(t)u \simeq t[u/x]$. This leads to the following definition:

▶ **Definition 18** (Quantitative Weak $\lambda$-Algebra)**.** *An applicative quantitative $\Sigma$-algebra $\mathcal{A} = (A, \Omega^{\mathcal{A}}, d^{\mathcal{A}})$ is called a* quantitative weak $\lambda$-algebra *if for all $w \in I^*$, $j \in I$, and $f \in \mathsf{Met}(A_w, A_j)$, the set $\Lambda(f) = \{g \in A_{w\to j} \mid \forall(x_1,\dots,x_k) \in A_w \quad g \cdot_A x_1 \cdot_A \dots \cdot_A x_k = f(x_1,\dots,x_k)\}$ is non-empty.*

▶ **Proposition 19.** *Any quantitative $\Sigma^{\mathbf{CL}}$-algebra satisfying $\mathcal{U}_{\mathbf{CL}}$ is a quantitative weak $\lambda$-algebra. Vice versa, any quantitative weak $\lambda$-algebra satisfies $\mathcal{U}_{\mathbf{CL}}$.*

▶ **Example 20.** We obtain a quantitative weak $\lambda$-algebra by letting $\mathcal{I} = (\mathcal{I}_i, \Omega^{\mathcal{I}}, d_i^{\mathcal{I}})$, where $\overline{r}^{\mathcal{I}} = r$, $\overline{f}^{\mathcal{I}} = f$, and $f \cdot^{\mathcal{I}} x = f(x)$. It is clear that $\mathcal{I} \vDash \mathcal{U}^{\mathcal{I}}_{\mathbf{CL}}$ (cf. Example 17).

Following [30], the condition from Definition 18 can be specified in categorial terms: a cartesian multicategory $\mathbb{C}$ is a model of **CL** precisely when for all objects $A, B$ of $\mathbb{C}$ there is an object $A \overset{\text{vw}}{\Rightarrow} B$ (called a *very weak exponential of $A$ and $B$*) together with a surjective natural transformation $\Phi : \mathbb{C}(_; A \overset{\text{vw}}{\Rightarrow} B) \to \mathbb{C}(_, A; B)$. When $\mathbb{C}$ is the multicategory $\mathsf{Met}^{\mathcal{A}}$, the conditions of Definition 18 imply that $A_{i\to j}$ is a very weak exponential of $A_i$ and $A_j$ in $\mathsf{Met}^{\mathcal{A}}$: a family of multiarrows $\mathbf{Ev}^{\mathcal{A}}_{w,i,j} : \mathsf{Met}^{\mathcal{A}}(A_w; A_{i\to j}) \Rightarrow \mathsf{Met}^{\mathcal{A}}(A_{w*i}; A_j)$, natural in $w$, is given by $\mathbf{Ev}^{\mathcal{A}}_{w,i,j}(f)(z,x) = f(z) \cdot^{A} x$, and the non-emptyness of the sets $\Lambda(f)$ corresponds to the surjectivity of this transformation.

Notice that $\mathsf{Met}$ itself admits very weak exponentials for all of its objects, *i.e.* it is a *very weak CCC* in the sense of [30], provided we endow $\mathsf{Met}(X,Y)$ with the metric $\Theta_{a,b}$ for metric spaces $(X,a)$ and $(Y,b)$, where for $f,g : X \to Y$ $\Theta_{a,b}(f,g)$ is 0 if $f = g$, and otherwise is $\sup\{b(f(x), g(y)) \mid x, y \in X\}$. Intuitively, when $f \neq g$, $\Theta_{a,b}(f,g)$ measures the diameter of the interval spanned by the image of both $f$ and $g$. However, the metric $\Theta_{a,b}$ is in general rather odd since the identity is an isolated point whenever $(X,a)$ is infinite and not trivial.

▶ **Example 21.** The constructions just sketched yields a different weak $\lambda$-algebra over the reals $\mathcal{I}_{\text{weak}} = (\mathcal{I}_i, \Omega^{\mathcal{I}}, d^{\text{weak}})$, where $d^{\text{weak}}$ is defined like $d^{\mathcal{I}}$ but for $d^{\text{weak}}_{i\to j} = \Theta_{d_i^{\text{weak}}, d_j^{\text{weak}}}$. Notice that we still have $\mathcal{I}_{\text{weak}} \vDash \mathcal{U}^{\mathcal{I}}_{\mathbf{CL}}$, since $\mathcal{I}$ and $\mathcal{I}_{\text{weak}}$ agree on distances of types $[a,b]$.

The result below adapts to the many-sorted case a similar result for one-sorted quantitative equational theories [29]. The proof is similar to that of Theorem 37, so we omit it.

▶ **Theorem 22** (Soundness and Completeness of Quantitative Weak $\lambda$-Theories)**.** *For any quantitative weak $\lambda$-theory $\mathcal{U}$ over $\Sigma^{\mathbf{CL}}$, $\Gamma \vdash \phi \in \mathcal{U}$ iff $\Gamma \vDash_{\mathcal{A}} \phi$ holds for any quantitative weak $\lambda$-algebra $\mathcal{A}$ such that $\mathcal{A} \vDash \mathcal{U}$.*

▶ Remark 23. Following Remark 13, in the case of partial ultra-metric spaces we will talk of *partial weak $\lambda$-theories* and *partial weak $\lambda$-algebras.*

## 5 Quantitative $\lambda$-Theories and Algebras

As we recalled, weak $\lambda$-theories do not fully capture the equational theory of the $\lambda$-calculus, as they fail to capture the so-called *$\xi$-rule* [5]. In our quantitative setting, this rule can be expressed as the inference $t \overset{j}{\simeq}_\epsilon u \vdash \lambda x.t \overset{i\to j}{\simeq}_\epsilon \lambda x.u$ *provided* the equation on the left of $\vdash$ is *locally universally quantified*: the righthand equation holds under the condition that, *for all possible value of $x$*, the lefthand equation holds. This kind of quantitative inferences

differ from those seen so far. The reason for this proviso is that it involves the higher-order operator $\lambda$, which "binds" the variable $x$. The example below shows that quantitative weak $\lambda$-algebras fail to capture this rule.

▶ **Example 24.** The $\xi$-rule fails in the weak $\lambda$-algebra $\mathcal{I}$: let $f, g : [0,b] \to [0, b+\epsilon]$ (where $b \in \mathbb{R}_{\geq 0}$ and $\epsilon \in \mathbb{Q}_{\geq 0}$) be, respectively, the identity function $f = \mathrm{id}$ and the function $g(x) = x + \epsilon$; for any $s \in [a,b]$, we then have $|f(s) - g(s)| \leq \epsilon$, which shows $\mathcal{I} \vDash \overline{f}x \overset{[a,b+\epsilon]}{\simeq_\epsilon} \overline{g}x$. However, since $d^{\mathcal{I}}_{[a,b]\to[a,b+\epsilon]}(f,g) = b + \epsilon$, we deduce $\mathcal{I} \nvDash \lambda x.\overline{f}x \overset{[a,b]\to[a,b+\epsilon]}{\simeq_\epsilon} \lambda x.\overline{g}x$.

In order to define quantitative $\lambda$-theories we could follow Curry [5] and "strengthen" the set of axioms, in fact mere equalities, satisfied by a $\Sigma^{\mathbf{CL}}$-algebra and essentially do away with the $\xi$-rule and all higher order features. The alternative, that we develop in this section, is to take abstraction and the $\xi$-rule as first class elements of our theories and algebras. This will require a number of generalizations of the original approach of [29].

At the level of syntax, the first step is to enrich the class of symbols with higher-order operators of the form $\lambda_i x$. The occurrence of the variable $x$ is part of the symbol $\lambda_i x$ itself.

▶ **Definition 25** ($\lambda$-Signature). *Given an applicative $T$-sorted signature $\Sigma$, let $\Sigma^\lambda$ be the applicative $T$-sorted signature further including the symbols $\lambda_i x \in \Sigma^\lambda_{j,i\to j}$, for all $x \in \mathrm{Var}_i$ and $i, j \in T$. The* $\lambda$-terms *are the elements of the free $\Sigma^\lambda$-algebra, $\mathbb{F}_\lambda(\mathrm{Var})$.*

Terms $\lambda_i x(t)$ will be denoted by $\lambda_i x.t$ or simply $\lambda x.t$. Free and bound variables, open and closed $\lambda$-terms are defined as usual. For a $\lambda$-term $t$, we denote by $\mathsf{fv}(t)$, $\mathsf{bd}(t)$, $\mathsf{var}(t)$ the sets of free, bound, and all variables in $t$, respectively. In order to simplify the notation we deal with bound variables by implementing directly Barendregt's "hygiene condition". For any function $f : \mathrm{Var} \to \mathbb{F}_\lambda(\mathrm{Var})$ there exists a function $f^\flat : \mathbb{F}_\lambda(\mathrm{Var}) \to \mathbb{F}_\lambda(\mathrm{Var})$ such that $f^\flat(t)$ corresponds to the substitution of $f(x)$ for $x$ in $t$, for any variable $x$ occurring free in $t$. Given pairwise disjoint variables $x_1, \dots, x_n$, with $x_j \in \mathrm{Var}_j$ and terms $t_1, \dots, t_n$, with $t_j \in \mathbb{F}_\lambda(\mathrm{Var})_j$, we indicate the "substitution" $(\mathrm{id}_{\vec{x},\vec{t}})^\flat(u)$ simply as $u[t_j/x_j]$.

In order to be able to express correctly the $\xi$-rule we generalize quantitative equations to expressions of the form $t \overset{Xi}{\simeq_\epsilon} u$, where $X$ indicates a finite set of variables which are intended to be "locally quantified" on the left of $\vdash$.

▶ **Definition 26** ($\Sigma^\lambda$-equation). *A* quantitative $\lambda$-equation *is an expression of the form $t \overset{X,i}{\simeq_\epsilon} s$, where $i \in I$, $t, s \in \Lambda_i$, $X \subseteq_{\mathsf{fin}} \mathrm{Var}$, $\epsilon \in \mathbb{Q}_{\geq 0}$. The set $X$ is the set of* locally quantified variables in the equation.

We let $\mathcal{V}(\Lambda)$ indicate the set of quantitative $\lambda$-equations.

▶ **Definition 27.** *A* consequence relation *on $\Lambda$ is a relation $\vdash \subseteq \wp(\mathcal{V}(\Lambda)) \times \mathcal{V}(\Lambda)$ closed under the rules (Cut)-(Nexp) from Def. 9 (with $t \overset{i}{\simeq_\epsilon} u$ everywhere replaced by $t \overset{Xi}{\simeq_\epsilon} u$), together with the following rules:*

***(Subst)*** *if $\Gamma \vdash t \overset{X,i}{\simeq_\epsilon} s$ and let $f$ be the identity on $X$ and, for all $x \in \mathrm{Var} \setminus X$, $\mathsf{fv}(f(x)) \cap \mathsf{bd}(t, s, \Gamma) = \emptyset$, then $\Gamma \vdash t \overset{X,i}{\simeq_\epsilon} s$ implies $f^\flat(\Gamma) \vdash f^\flat(t) \overset{X,i}{\simeq_\epsilon} f^\flat(s)$;*

***(Abstraction)*** *if $X \subseteq X'$ and $\mathsf{fv}(t,s) \cap X' = \emptyset$, then $\{t \overset{X,i}{\simeq_\epsilon} s\} \vdash t \overset{X',i}{\simeq_\epsilon} s$;*

***(Concretion)*** *if $X' \subseteq X$ then $\{t \overset{X,i}{\simeq_\epsilon} s\} \vdash t \overset{X',i}{\simeq_\epsilon} s$;*

*We call $\mathcal{U}_\lambda$ the quantitative theory generated by the axioms below apart from ($\eta$) and we denote by $\vdash^{\boldsymbol{\lambda}}$ the corresponding consequence relation, and $\mathcal{U}_{\lambda\eta}$ the quantitative theory generated by all the axioms below, including ($\eta$), with consequence relation $\vdash^{\boldsymbol{\lambda\eta}}$:*

**($\alpha$)** *if* $x, y \in \mathrm{Var}_i$ *and* $y \notin \mathsf{var}(\lambda_i x.t)$*, then* $\emptyset \vdash \lambda x.t \overset{X,i}{\simeq_0} \lambda_i y.t[y/x]$.
**($\xi$)** *if* $x \in X$*, then* $t \overset{X,j}{\simeq_\epsilon} u \vdash \lambda_i x.t \overset{X,i\to j}{\simeq_\epsilon} \lambda_i x.u$;
**($\beta$)** *if* $(\lambda_i x.t)u \in \Lambda_j$, $\mathsf{fv}(u) \cap \mathsf{bd}(t) = \emptyset$*, then* $\emptyset \vdash (\lambda_i x.t)u \overset{X,j}{\simeq_0} t[u/x]$.
**($\eta$)** *if* $\lambda_i x.(tx) \in \Lambda_{i\to j}$, $x \notin \mathsf{fv}(t)$*, then* $\vdash t \overset{X,i\to j}{\simeq_0} \lambda_i x.(tx)$.

*Any theory including* $\mathcal{U}_\lambda$ *(*$\mathcal{U}_{\lambda\eta}$*) is called a* quantitative (extensional) $\lambda$-theory*.*

▶ **Example 28.** Consider the $\lambda$-signature $\mathcal{I}(\Sigma)^\lambda$ (cf. Example 17). Let $\mathcal{U}^{\mathcal{I}}_{\lambda\eta}$ be the extensional $\lambda$-theory obtained by enriching $\mathcal{U}_{\lambda\eta}$ with all real-valued axioms as in Example 17.

We now introduce a class of applicative algebras suitable to account for abstraction operators. This is done by requiring the existence of suitable "closing maps" that send a closed $\lambda$-term of the form $\lambda_{i_1} x_1 \dots . \lambda_{i_n} x_n.t$ onto some point of $A_{i_1 \to \dots \to i_n \to j}$.

Given any $T$-index set $A$, extend the definition of $\Sigma^\lambda$ to $\Sigma^{\lambda,A}$ so as to contain as 0-ary constructors all elements in $A$ and correspondingly the notion of $\mathbb{F}_{\lambda,A}(\mathrm{Var})$.

▶ **Definition 29.** *$A$* quantitative applicative $\lambda$-algebra *is a structure* $\mathcal{A} = (A, \Omega^{\mathcal{A}}, \Lambda^{\mathcal{A}}, d^{\mathcal{A}})$*, where* $(A, \Omega^{\mathcal{A}}, d^{\mathcal{A}})$ *is a quantitative applicative algebra and* $\Lambda^{\mathcal{A}}_{w,j} : (\mathbb{F}_{\lambda,A}(\mathrm{Var}))^0_{w\to j} \to A_{i\to j}$*. We call* applicative $\lambda$-algebra *the structure* $\mathcal{A} = (A, \Omega^{\mathcal{A}}, \Lambda^{\mathcal{A}})$ *without the metric.*

The functions $\Lambda^{\mathcal{A}}_{i,j} : (\mathbb{F}_{\lambda,A}(\mathrm{Var}))^0_{i\to j} \to A_{i\to j}$ are intended to define a choice in the set $\Lambda$ of Definition 18. This will be apparent in view of Definitions 30, 31, 32 below, which will enforce that, in suitable structures, the interpretations of the terms $\mathbb{F}_{\lambda,A}(\mathrm{Var}))^0_{i\to j}$ become essentially the domain of $\Lambda$ in Definition 18. We point out that a slight modification of these definitions would permit to recover precisely the categorically weaker notion of Quantitative Weak $\lambda$-algebra of Definition 18.

▶ **Proposition 30** (Interpretation)**.** *Let $\mathcal{A}$ be a quantitative applicative $\lambda$-algebra, and* $\rho : \mathrm{Var} \to A$*. Then there exists a function* $\rho^\natural : \mathbb{F}_\lambda(\mathrm{Var}) \to A$*, where* $\rho^\natural(t)$ *is defined by cases as* $\rho^\natural(x) = \rho(x)$, $\rho^\natural(t_1 \cdot t_2) = \rho^\natural(t_1) \cdot_A \rho^\natural(t_2)$*, and* $\rho^\natural(\lambda_i x.t) = \Lambda^{\mathcal{A}}_{w*i,j}(\lambda \vec{y} \lambda_i x.t) \cdot_A \overrightarrow{\rho^\natural(y)}$*, where* $\lambda \vec{y} \lambda_i x.t$ *is the closure of the term* $\lambda_i x.t$ *w.r.t. its free variables* $\vec{y}$ *of types* $w$*.*

To define higher-order structure for a quantitative applicative $\lambda$-algebras $\mathcal{A}$ it is useful to define the multicategory generated by $\mathcal{A}$:

▶ **Definition 31** (Representable Functions)**.** *For any quantitative applicative $\lambda$-algebra $\mathcal{A}$,* $\mathsf{Met}^{\mathcal{A}}$ *is the multicategory with objects the metric spaces* $(A_i, d^{\mathcal{A}}_i)$*, and where, for* $w = i_1 \dots i_k$*,* $\mathsf{Met}^{\mathcal{A}}(A_{i_1}, \dots, A_{i_k}; A_i)$ *(abbreviated as* $\mathsf{Met}^{\mathcal{A}}(A_w; A_i)$*) is the set of* $f \in \mathsf{Met}(A_w, A_i)$ *such that for some* $t_f \in \mathbb{F}_{\lambda,A}(\mathrm{Var})^0_{w\to i}$*,* $f(a_1, \dots, a_n) = \Lambda^{\mathcal{A}}_{w,i}(t_f) \cdot^{\mathcal{A}} \vec{a}$*.*

Notice that the function $\Lambda^{\mathcal{A}}$ yields a family of maps $\mathbf{\Lambda}^{\mathcal{A}}_{w*i,j} : \mathsf{Met}^{\mathcal{A}}(A_{w*i}; A_j) \to \mathsf{Met}^{\mathcal{A}}(A_w; A_{i\to j})$, given by $\mathbf{\Lambda}_{w*i,k}(h)(a)(b) = \Lambda^{\mathcal{A}}_{w*i,j}(t_h) \cdot^{\mathcal{A}} \langle a, b \rangle$.

While cartesian closed (multi)categories are the algebras for extensional $\lambda$-theories, an algebra for a $\lambda$-theory is a cartesian multicategory in which for all objects $A, B$ there is an object $A \overset{\mathrm{w}}{\Rightarrow} B$ (called a *weak exponential*, [30]) together with a natural *retraction* $\mathbb{C}(_, A; B) \Rightarrow \mathbb{C}(_; A \overset{\mathrm{w}}{\Rightarrow} B)$. To account for the quantitative $\xi$-rule, this picture must be slightly adapted, by requiring the maps forming the retraction to be also *non-expansive*. This leads to the following definition:

▶ **Definition 32** (Quantitative $\lambda$-Algebra)**.** *For any quantitative applicative $\lambda$-algebra* $\mathcal{A} = (A, \Omega^{\mathcal{A}}, \Lambda^{\mathcal{A}}, d^{\mathcal{A}})$*, $\mathcal{A}$ is a* quantitative (extensional) $\lambda$-algebra *if the maps* $\mathbf{\Lambda}^{\mathcal{A}}_{w*i,j}, \mathbf{Ev}^{\mathcal{A}}_{w,i,j}$ *form a family of retractions (resp. isomorphisms) natural in $w$ and non-expansive (with respect to the pre-metrics* $\Xi_{d^{\mathcal{A}}_w, d^{\mathcal{A}}_j}$ *over* $\mathsf{Met}^{\mathcal{A}}(A_w; A_j)$*).*

The definition above can be expressed in more abstract terms using the language of *enriched* categories: the multicategory $\mathsf{Met}^{\mathcal{A}}$ is enriched over the cartesian closed category of pre-metric spaces and non-expansive functions, where $\mathsf{Met}^{\mathcal{A}}(A_w; A_i)$ is endowed with the pre-metric $\Xi_{d_w^{\mathcal{A}}, d_i^{\mathcal{A}}}$. Then $\mathcal{A}$ is a quantitative (resp. extensional) $\lambda$-algebra when the maps $\mathbf{\Lambda}^{\mathcal{A}}_{w*i,j}, \mathbf{Ev}^{\mathcal{A}}_{w,i,j}$ form an enriched natural retraction (resp. isomorphism) from $\mathsf{Met}^{\mathcal{A}}(A_{w*i}; A_j)$ to $\mathsf{Met}^{\mathcal{A}}(A_w; A_{i\to j})$. Notice that this condition implies that the pre-metrics $\Xi_{d_w^{\mathcal{A}}, d_i^{\mathcal{A}}}$ are indeed metrics.

▶ **Example 33.** $\mathcal{I}$ becomes a quantitative $\lambda$-algebra by defining $\Lambda^{\mathcal{I}}$ inductively on $\mathbb{F}_{\lambda,\mathcal{I}}(\mathrm{Var})^0$, exploiting the cartesian closed structure of the subcategory of $\mathsf{Met}$ formed by the spaces $\mathcal{I}_i$.

Let us now show how quantitative $\lambda$-algebras are captured by quantitative $\lambda$-theories.

▶ **Definition 34.** *Let $\mathcal{A} = (A, \Omega^{\mathcal{A}}, \Lambda^{\mathcal{A}}, d^{\mathcal{A}})$ be a quantitative applicative $\lambda$-algebra. For any $\rho : \mathrm{Var} \to A$, we say that $\mathcal{A}$ satisfies a quantitative equation $\phi = t \overset{X,i}{\simeq_\epsilon} u$* relative to *$\rho$, (denoted $\vDash^{\rho}_{\mathcal{A}} \phi$), where $X = \{x_1, \dots, x_n\}$, with $x_1 \in A_{i_1}, \dots, x_n \in A_{i_n}$, when for all $a_1, b_1 \in A_{i_1}, \dots, a_n, b_n \in A_{i_n}$, the following condition holds:*

$$d_i^{\mathcal{A}}(\rho^{\natural}_{\vec{x},\vec{a}}(t), \rho^{\natural}_{\vec{x},\vec{b}}(u)) \leq \max\{\epsilon, d_{i_1}^{\mathcal{A}}(a_1, b_1), \dots, d_{i_n}^{\mathcal{A}}(a_n, b_n)\}$$

*We say that $\mathcal{A}$ satisfies a quantitative $\lambda$-inference $\Gamma \vdash \phi$ (denoted $\Gamma \vDash_{\mathcal{A}} \phi$) if for all $\rho : \mathrm{Var} \to A$, if $\vDash^{\rho}_{\mathcal{A}} \psi$ holds for all $\psi \in \Gamma$, then $\vDash^{\rho}_{\mathcal{A}} \phi$ also holds. $\mathcal{A}$ satisfies a quantitative $\lambda$-theory $\mathcal{U}$ (denoted $\mathcal{A} \vDash \mathcal{U}$) if it satisfies all the inferences in $\mathcal{U}$.*

Definition 34 of satisfiability is admittedly more complex than Definition 12. Yet, this is the price one has to pay in order to be able to express the quantitative $\xi$-rule. Indeed, the definition of $\vDash^{\rho}_{\mathcal{A}} \phi$ treats "locally quantified" variables by applying a condition reminiscent of the metrics $\Xi$ from Section 2: for all locally quantified variables $\vec{x}$ in $\phi = t \overset{X,i}{\simeq_\epsilon} u$, when the $\vec{x}$ are replaced in $t$ and $u$ by *different* points $\vec{a}, \vec{b}$, the distance between the resulting terms must be bounded by either $\epsilon$ or any of the $d^{\mathcal{A}}_{i_j}(a_j, b_j)$. This ensures that, whenever $\vDash_{\mathcal{A}} t \overset{\{x\},j}{\simeq_\epsilon} u$ is satisfied, we can conclude $\Xi(\lambda x.t, \lambda x.u) \leq \epsilon$, as the $\xi$-rule requires.

▶ **Example 35.** Def. 34 solves the problem from Example 24: with $r = 0$ and $s = \epsilon$, from the fact that $|f(s) - g(r)| = 2\epsilon > \max\{|r - s|, \epsilon\}$, it follows that $\mathcal{I} \nvDash \overline{f}x \overset{X,j}{\simeq_\epsilon} \overline{g}x$, hence blocking the counter-example to the $\xi$-rule. Rather, it holds that $\mathcal{I} \vDash \mathcal{U}^{\mathcal{I}}_{\lambda\eta}$ (cf. Example 28).

▶ **Proposition 36.** *A quantitative applicative $\lambda$-algebra is a quantitative $\lambda$-algebra (resp. a quantitative extensional $\lambda$-algebra) iff it satisfies $U_\lambda$ (resp. $U_{\lambda\eta}$).*

From the argument of the proposition above one can also deduce that a quantitative applicative $\lambda$-algebra is a weak $\lambda$-algebra iff it satisfies $(\alpha)$ and $(\beta)$, and is an (extensional) $\lambda$-algebra iff it furthermore satisfies $(\xi)$ (and $(\eta)$).

We conclude this section by showing soundness and completeness of quantitative (extensional) $\lambda$-theories. The proof is based on the construction of a "quantitative term model".

▶ **Theorem 37** (Soundness and Completeness of Quantitative $\lambda$-theories)**.** *Let $\mathcal{U}$ be a quantitative $\lambda$-theory (resp. a quantitative extensional $\lambda$-theory) over $\Sigma^{\lambda}$. Then $\Gamma \vdash^{\boldsymbol{\lambda}} \phi \in \mathcal{U}$ (resp. $\Gamma \vdash^{\boldsymbol{\lambda\eta}} \phi \in \mathcal{U}$) iff $\Gamma \vDash_{\mathcal{A}} \phi$ holds for any quantitative $\Sigma^{\lambda}$-algebra (resp. quantitative extensional $\Sigma^{\lambda}$-algebra) $\mathcal{A}$ such that $\mathcal{A} \vDash \mathcal{U}$.*

▶ Remark 38. Also in this case the whole construction scales to the case of partial ultra-metric spaces. Following Remark 23, we will speak of *partial $\lambda$-theories* and *partial $\lambda$-algebras.*

## 6 Metric Constraints

In this section we take a closer look at the several obstacles one might face when looking for higher-order quantitative algebras. First, as seen in Section 2, in higher-order types the *unique* distance, $\Xi$, making both application and abstraction non-expansive operations might not be a metric. Moreover, even if such a metric exists, several conditions might lead higher-order distances to be trivial (i.e. discrete), or have plenty of isolated points. But discrete metrics and isolated points convey no more information than equivalences, while one of the main reasons to look for semantics of program distances is to be able to compare informatively programs which are not equivalent. Despite what look like strong limitations, we conclude this section by presenting a few examples of non-discrete quantitative $\lambda$-algebras.

**Existence of Exponential Objects.** Given metric spaces $(X, a)$ and $(Y, b)$, if $(Y, b)$ is ultra-metric, then $\Xi_{a,b} = \Phi_{a,b}$ is always a metric, which means that $\mathsf{Met}(X, Y)$ is their exponential object in $\mathsf{Met}$. When $(Y, b)$ is not ultra-metric, condition $(*)$ from Theorem 2 provides a useful *sufficient* criterion to check if $\Xi_{a,b}$ is a metric (and thus, if some candidate quantitative applicative $\lambda$-algebra $\mathcal{A}$ is a quantitative $\lambda$-algebra). We will now show that, under very mild hypotheses, the validity of $(*)$ is also *necessary* for $\mathcal{A}$ to be a quantitative $\lambda$-algebra.

Let a quantitative applicative $\lambda$-algebra $\mathcal{A}$ be *observationally complete* when it contains the metric space $(\mathbb{R}^{\infty}_{\geq 0}, |\cdot - \cdot|)$ and for all sort $i$, $\mathsf{Met}^{\mathcal{A}}(A_i; \mathbb{R}^{\infty}_{\geq 0}) \simeq \mathsf{Met}(A_i, \mathbb{R}^{\infty}_{\geq 0})$. In other words, $\mathcal{A}$ contains *all* observations on $A_i$ with target $\mathbb{R}^{\infty}_{\geq 0}$. Moreover, let a *quantitative $\lambda$-pre-algebra* be as a quantitative $\lambda$-algebra $\mathcal{A}$, but where the $d_i^{\mathcal{A}}$ need only be pre-metrics. Given a quantitative $\lambda$-pre-algebra $\mathcal{A}$, let $\mathcal{A}^*$ indicate the restriction of $\mathcal{A}$ to those sorts $i$ for which $d_i^{\mathcal{A}}$ is a metric (i.e. it also satisfies (trans)).

▶ **Proposition 39.** *Let $\mathcal{A}$ be an observationally complete quantitative extensional $\lambda$-pre-algebra. For any $A$ in $\mathcal{A}^*$, $A$ is exponentiable in $\mathsf{Met}^{\mathcal{A}^*}$ iff for all $\alpha, \beta \in \mathrm{Im}(d^{\mathcal{A}})$ and $x_0, x_2 \in X$ with $a(x_0, x_2) = \alpha + \beta$, condition $(*)$ holds.*

Proposition 39 has a positive side: it provides a sufficient condition for exponentiability which is slightly weaker than Theorem 2, as $(*)$ needs only hold for distances $\alpha, \beta$ in the *image* of the distance functions $d^{\mathcal{A}}$ of the pre-algebra. Notice that, if the $d^{\mathcal{A}}$ are discrete, condition $(*)$ trivially holds. On the other hand, Proposition 39 has a negative side: if condition $(*)$ fails (i.e. some space $A$ does not contain "enough points"), then $\mathcal{A}$ fails to be a quantitative $\lambda$-algebra. For instance, no algebra containing $\mathbb{N}$, with the metric inherited from $\mathbb{R}$, as one of its objects, can be a $\lambda$-algebra.

**Existence of Compact Algebras.** We have the following negative result.

▶ **Proposition 40.** *There are no non-trivial one-sorted weak quantitative $\lambda$-algebras in* $\mathsf{Met}$ *which are compact.*

By contrast, in the multi-sorted case, compact $\lambda$-algebras do exist, e.g. take the restriction of the quantitative $\lambda$-algebra $\mathcal{I}$ to compact intervals $[a, b]$, i.e. with $a, b < \infty$, or simply the full type structure on a finite base set.

**Distances and Observational Equivalence.** The next two results relate distances in quantitative $\lambda$-theories with observational equivalence for the associated $\lambda$-theory, clearly indicating the (limited) extent to which a metric can deviate from being discrete on *pure* closed $\lambda$-terms. We recall that *pure* means that no constants appear in the syntax, or categorically, that the $\Sigma^{\lambda}$-signature has only the $\cdot$ symbols.

▶ **Proposition 41.** *In a quantitative $\lambda$-algebra $A$,* i.e. *a model of the simply typed $\lambda$-calculus, terms which are not equated in the maximal theory are all at the same distance from one another. Moreover each $A_i$ is a bounded pseudo-metric space.*

The maximal non-trivial theory of the *pure* simply typed $\lambda$-calculus is the theory $FTS$ of the *full type structure* over a two-element base set [7]. Proposition 41 implies then that any quantitative $\lambda$-algebra for $FTS$ is discrete. We recall that "pure" means that no constants appear in the syntax. Next, we consider the untyped $\lambda$-calculus:

▶ **Proposition 42.** *In a non-trivial weak quantitative $\lambda$-algebra $A$, the maximal distance between any two points is bounded by $d([\![\mathsf{K}]\!], [\![\mathsf{K}(\mathsf{SKK})]\!])$. Hence all pairs of terms which can be applied, by a given term, on $[\![\mathsf{K}]\!]$ and $[\![\mathsf{K}(\mathsf{SKK})]\!]$ respectively, are that distance apart. Moreover, if $A$ is a non-trivial quantitative $\lambda$-algebra then for any two solvable terms, $t$ and $s$, which are not equated in the maximal theory $\mathcal{H}^*$ (see [5]) and $\mathsf{Y}$, fixed-point combinator we have $d([\![t]\!], \mathsf{YK}) = d([\![s]\!], \mathsf{YK})$. If the distance is ultra-metric we have also $d([\![t]\!], \mathsf{YK}) = d([\![t]\!], [\![s]\!])$. In any case, if the theory equates all unsolvable terms then $d([\![t]\!], \mathsf{YK}) \leq d([\![t]\!], [\![s]\!])$.*

As a consequence of Böhm Theorem (see [5]), Proposition 42 implies that any quantitative $\lambda$-algebra for the *pure* untyped $\lambda$-calculus is discrete over $\beta\eta$-normal forms.

**Positive Examples.** The above limiting results apply only to terms which are not equated in the maximal theories of the $\lambda$-calculus, either typed or untyped ([5, 7]). Clearly these terms are significant computationally, and this is the bad news, but these terms are rather special and hence Propositions 41 and 42 have only a limited negative impact, and this is the good news. For instance, in the maximal theory of the simply typed $\lambda$-calculus Church, numerals are equated up to parity, so Proposition 41 does not have any bearing on the mutual distance of two different even, or two different odd, numerals. Indeed, rather intriguing distances in quantitative $\lambda$-algebras do exist, even in the category of complete (not necessarily ultra-) metric spaces and non-expansive functions, as the following examples show.

Any complete partial order model of Combinatory Logic, and hence in particular of $\lambda$-calculus (*e.g.* any Scott's inverse limit $\mathrm{D}_\infty$ model, [5]), can be endowed with the metric

$$d(d_1, d_2) = \begin{cases} 0 & \text{if } d_1 = d_2 \\ 1/2 & \text{if } d_1 \text{ and } d_2 \text{ have an upper bound} \\ 1 & \text{otherwise .} \end{cases}$$

One can check that application is non-expansive, and that the space is complete; moreover the space of representable functions (*i.e.* functions determined by the elements of the model), endowed with the supremum metric, is isometrically embedded in the space. Alternatively, one can consider the term model of the simply typed $\lambda$-calculus with a base constant $\bot$. By strong normalization, it consists of the $\beta\eta$-normal forms. Let $\sqsubseteq$ be the order relation defined on normal forms of the same type by $\lambda\vec{x}.\ \bot \sqsubseteq \lambda\vec{x}.t$ and $\lambda\vec{x}.x_i t_1 \dots t_k \sqsubseteq \lambda\vec{x}.x_i u_1 \dots u_k$, if $t_i \sqsubseteq u_i$ for all $i = 1, \dots, k$ (corresponding to the natural order relation on Böhm trees, see [5]). The set of $\beta\eta$-normal forms can be endowed with a notion of distance by putting, for all type $\sigma \in T$ and $t, u$ terms of type $\sigma$, $d_\sigma(t, u)$ be 0 if $t = u$, 1/2 if $t$ and $u$ have an upper bound, and 1 otherwise.

Yet other distances can be given on the term model of the simply typed $\lambda$-calculus by putting $d_\sigma(t, u)$ be 0 if $t =_{\beta\eta} u$ and otherwise $1/N$, where $N = \max\{\ n \mid [\![t]\!] = [\![u]\!]$ in the full type hierarchy over $n$ points $\}$.

# 7 Partial Quantitative $\lambda$-Algebras

In this section we discuss *partial metrics*, and the natural generalization of quantitative $\lambda$-algebras to *partial quantitative $\lambda$-algebras*. In particular we define two non-trivial such algebras for the simply typed $\lambda$-calculus. The first $\lambda$-algebra that we consider is defined on the term model of $\beta\eta$-normal forms of the simply typed $\lambda$-calculus with a constant $\bot$ of base type. The latter is defined within a $D_\infty$ $\lambda$-model *à la* Scott. In both cases we define an ultra-metric distance using a suitable notion of *term approximants*.

**The Partial $\lambda$-Algebra of the Term Model.** Let $T$ be the set of simple types built over the base type $o$, and let $\sigma, \tau$ range over $T$. The $\beta\eta$-normal forms of the simply typed $\lambda$-calculus with constant $\bot$ of type $o$ can be endowed with a structure of applicative $\lambda$-algebra:

▶ **Proposition 43.** *Let $\mathcal{NF} = (NF, \Omega^{NF}, \Lambda^{NF})$ be the structure where:*
- *$NF$ is the $T$-indexed set of typed $\beta\eta$-normal forms with constant $\bot$ of type $o$,*
- *for all $\sigma, \tau$, $\cdot_{\sigma,\tau} : NF_{\sigma\to\tau} \times NF_\sigma \to NF_\tau$ is defined by $t \cdot_{\sigma,\tau} s = [ts]_{\beta\eta}$,*
- *$\Lambda^{NF}_{\sigma,\tau} : \mathbf{\Lambda}^0_{\sigma\to\tau} \to NF_{\sigma\to\tau}$ is defined by: $\Lambda^{NF}_{\sigma,\tau}(\lambda x.t) = [\lambda x.t]_{\beta\eta}$,*

*where $[t]_{\beta\eta}$ denotes the $\beta\eta$-normal form of $t$. Then $\mathcal{NF}$ is an applicative $\lambda$-algebra.*

The signature of this algebra can be enriched with *projection operators* providing the approximants of a given normal form. Intuitively, the $n^{\text{th}}$ approximant of a normal form is the term whose Böhm tree [5] is obtained by cutting all branches at depth $n$ and by labelling leaves at level $n$ of type $\sigma_1 \to \ldots \to \sigma_m \to o$ by the term $\lambda x_1 \ldots x_m.\ \bot$. More precisely:

▶ **Definition 44.** *For all $\sigma \in T$ and for all $n \in \mathbb{N}$, $\pi^n_\sigma : NF_\sigma \to NF_\sigma$ is defined by induction on $n$ as follows: for all $\lambda x_1 \ldots x_m.x_i t_1 \ldots t_k \in NF_\sigma$, $\pi^\sigma_0(t) = \lambda x_1 \ldots x_m.\bot$ and $\pi^\sigma_{n+1}(t) = \lambda x_1 \ldots x_m.x_i(t_1)_n \ldots (t_k)_n$.*

In the sequel, we will denote the approximant $\pi^n_\sigma(t)$ simply by $t_n$.

▶ **Definition 45** (Distance on Normal Forms)**.** *We define a family of functions $d^{NF} = \{d^{NF}_\sigma\}_\sigma$, where $d^{NF}_\sigma : NF_\sigma \times NF_\sigma \to \mathbb{R}_{\geq 0}$ is defined inductively by*

$$d^{NF}_o(t,s) = \begin{cases} 0 & \text{if } t = s \\ 1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad d^{NF}_{\sigma\to\tau}(t,t') = \frac{1}{2^m}$$

*where $m$ is the largest $n \in \mathbb{N}$, if it exists, such that*
**(1)** *$t_n = t'_n$,*
**(2)** *$\forall s, s' \in NF_\sigma.\ (d^{NF}_\sigma(s,s') \leq \frac{1}{2^n} \implies d^{NF}_\tau(ts,ts') \leq \frac{1}{2^n})$;*
*if such a maximal $n$ does not exist, then $d^{NF}_{\sigma\to\tau}(t,t')$ is set to $0$.*

▶ **Lemma 46.** *For all $\sigma, \tau$, $(NF_\sigma, d^{NF}_\sigma)$ is a partial ultra-metric space and $\cdot_{\sigma,\tau}$ is non-expansive.*

The fact that $\cdot_{\sigma,\tau}$ is non-expansive follows immediately from the definition of $d^{NF}_\sigma$.

▶ Remark 47. Notice that $d^{NF}$ is not reflexive. *E.g.*, let $u = \lambda x.xI$ of appropriate type $\sigma_1 \to \sigma_2$, $t = \lambda x.x(xt')$ and $s = \lambda x.x(xs')$ of type $\sigma_1$, and $t', s'$ of appropriate type $\tau$ such that $d^{NF}_\tau(t',s') = 1$. Then $d^{NF}_{\sigma_1}(t,s) = \frac{1}{2}$, but $d_{\sigma_2}(ut, us) = 1$, *i.e.* $d_{\sigma_1\to\sigma_2}(u,u) = 1$.

From the definition of $d^{NF}$, it immediately follows that application on normal forms is a non-expansive operator. Hence we have:

▶ **Proposition 48.** *$\mathcal{NF} = (NF, \Omega^{NF}, \Lambda^{NF}, d^{NF})$ is a partial quantitative extensional $\lambda$-algebra.*

▶ Remark 49. If we drop condition 2 in Definition 45, then we get an ultra-metric (reflexivity holds), however application is expansive. Namely, let $t = \lambda x_1 x_2.x_1(x_2 t')$ and $s = \lambda x_1 x_2.x_1(x_2 s')$ of appropriate types $\sigma_1$ such that $d^{NF}_{\sigma'}(t', s') = 1$. Then $d^{NF}_{\sigma_1}(t, s) = 1/4$, but for $u = \lambda x.xII$ of type $\sigma_1 \to \sigma_2$, we get $d^{NF}_{\sigma_2}(ut, us) = 1$. Notice that the above terms can be taken to be affine (similar counterexamples can be built also in the purely linear case).

**The Partial $\lambda$-Algebra of $D_\infty$.** Any inverse limit domain model *à la* Scott of $\lambda$-calculus yields an applicative $\lambda$-algebra. On such models a notion of approximant naturally arises, by considering for any given element of the domain its projections on the domains of the inverse limit construction. This leads to the following definition:

▶ **Definition 50.**

**(i)** *Let $D_\infty = \bigsqcup_n D_n$ be an inverse limit domain model* à la *Scott. For all $a \in D_\infty$ we define the $n^{th}$ approximant of $a$, $a_n$, as the projection of $a$ into $D_n$.*

**(ii)** *Let $D$ be the $T$-indexed set $\{D_\sigma\}_\sigma$, where, for all $\sigma$, $D_\sigma = D_\infty$.*

**(iii)** *Let $d^{\mathcal{D}}_\sigma : D_\sigma \times D_\sigma \to \mathbb{R}_{\geq 0}$ be the distance function defined by induction on types by*

$$d^{\mathcal{D}}_o(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad d^{\mathcal{D}}_{\sigma\to\tau}(a, b) = \tfrac{1}{2^m}$$

*where $m$ is the maximal $n \in \mathbb{N}$, if it exists, such that*

**(1)** *$a_n = b_n$,*

**(2)** *$\forall c, d \in D_\sigma.\ (d^{\mathcal{D}}_\sigma(c, d) \leq \frac{1}{2^n} \implies d^{\mathcal{D}}_\tau(ac, bd) \leq \frac{1}{2^n})$;*

*if such a maximal $n$ does not exists, then $d^{\mathcal{D}}_{\sigma\to\tau}(a, b) = 0$.*

▶ **Lemma 51.** *For all $\sigma, \tau$, $(D_\sigma, d^{\mathcal{D}}_\sigma)$ is a partial ultra-metric space and $\cdot_{\sigma,\tau}$ is non-expansive.*

▶ **Proposition 52.** *Let $\mathcal{D} = (D, \Omega^{\mathcal{D}}, \Lambda^{\mathcal{D}}, d^{\mathcal{D}})$ be the structure where the functions $\Lambda^{\mathcal{D}}_{\sigma,\tau}$ are the interpretations of closed typed $\lambda$-terms on $D_\infty$. Then $\mathcal{D}$ is a partial quantitative extensional $\lambda$-algebra.*

**Partial $\lambda$-Algebras with Approximants.** The two examples above of partial quantitative $\lambda$-algebras can be viewed as special cases of a general construction, which can be carried out on any applicative algebra which includes projection operators. Namely, using the system of approximants given by the projection operators, one can endow the algebra with an ultra-metric, getting a quantitative applicative algebra. If moreover the algebra satisfies the $(\beta)$-rule, then it is a quantitative (weak) $\lambda$-algebra.

▶ **Definition 53** (Applicative Algebra with Approximants)**.** *An applicative algebra $\mathcal{A} = (A, \Omega^{\mathcal{A}})$ has* approximants *if it includes projection operators $\pi^n_\sigma : A_\sigma \to A_\sigma$, for all $\sigma \in T$ and $n \in \mathbf{N}$, satisfying the following property: for all $n \in \mathbb{N}$, for all $a, b \in A_\sigma$, $\pi^{n+1}_\sigma(a) = \pi^{n+1}_\sigma(b)$ implies $\pi^n_\sigma(a) = \pi^n_\sigma(b)$.*

▶ **Proposition 54.** *Let $\mathcal{A} = (A, \Omega^{\mathcal{A}})$ be an applicative $\lambda$-algebra with approximants. Let $d^{\mathcal{A}}_\sigma : A_\sigma \times A_\sigma \to \mathbb{R}_{\geq 0}$ be the family of functions defined by induction on types as follows:*

$$d^{\mathcal{A}}_o(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \qquad \text{and} \qquad d^{\mathcal{A}}_{\sigma\to\tau}(a, b) = \frac{1}{2^m}$$

*where $m$ is the maximal $n \in \mathbb{N}$, if it exists, such that*

**(1)** $\pi^n_{\sigma\to\tau}(a) = \pi^n_{\sigma\to\tau}(b)$,
**(2)** $\forall c, d \in A_\sigma.\ (d^A_\sigma(c,d) \leq \frac{1}{2^n} \implies d^A_\tau(ac, bd) \leq \frac{1}{2^n})$;
*if the maximal $n$ does not exist, then $d^A_{\sigma\to\tau}(a,b) = 0$. Then $(A, \Omega^{\mathcal{A}}, d^{\mathcal{A}})$ is a quantitative applicative algebra.*

## 8 Approximate Quantitative Algebras

As we have seen, finding non-discrete quantitative (weak) $\lambda$-algebras is difficult. One difficulty arises from the non-expansiveness requirement on application. In Section 7 we have shown how to define non-trivial ultra-metric quantitative $\lambda$-algebras, still maintaining the non-expansiveness requirement for application, but at the price of the partiality of the metric. Here we present a different approach: we relax rule (NExp) for application, so as to get quantitative $\lambda$-algebras with full pseudo-metric distances. Namely, we introduce the notion of *approximate applicative algebra*: this amounts to an applicative algebra with approximants (see Definition 53 above), and operators $\{\cdot^n\}_{n\in\mathbb{N}}$ approximating application. Projection operators immediately induce an ultra-metric on the algebra, by just considering condition 1 in Proposition 54 (and dropping condition 2). In general, application is expansive w.r.t. this metric (see Remark 49). However, the milder *uniform non-expansiveness* condition for approximant operators is satisfied in many cases, including the term algebra of normal forms and the $D_\infty$ model of Section 7. This approach is quite general, since it works both for the typed and the untyped $\lambda$-calculus.

▶ **Lemma 55.** *Let $\mathcal{A} = (A, \Omega^{\mathcal{A}})$ be an applicative algebra with approximants. Let $e^{\mathcal{A}}_\sigma : A_\sigma \times A_\sigma \to \mathbb{R}_{\geq 0}$ be the family of functions defined by: $e^{\mathcal{A}}_\sigma(a,b) = \frac{1}{2^m}$ , where $m$ is the maximal $n \in \mathbb{N}$, if it exists, such that $a_n = b_n$, otherwise we put $e^{\mathcal{A}}_\sigma(a,b) = 0$. Then for all $\sigma$ $(A, e^{\mathcal{A}}_\sigma)$ is an ultra-metric space.*

▶ **Definition 56** (Approximate Quantitative Algebra)**.**

**(i)** *An* approximate algebra $\mathcal{A} = (A, \Omega^{\mathcal{A}})$ *is an applicative algebra with approximants whose signature includes also a family of operators $\cdot^n_{\sigma,\tau} : A_{\sigma\to\tau} \times A_\sigma \to A_\tau$, for all $\sigma, \tau \in T$ and $n \in \mathbf{N}$ (the operators $\cdot^n_{\sigma,\tau}$ will be simply denoted by $\cdot^n$).*

**(ii)** *An* approximate quantitative algebra $\mathcal{A} = (A, \Omega^{\mathcal{A}}, e^{\mathcal{A}})$ *is an approximate algebra where the operators $\cdot^n_{\sigma,\tau}$ satisfy the following conditions:*

  **(1)** *for all $a \in A_{\sigma\to\tau}$, $b \in A_\sigma$, $n \in \mathbf{N}$, $e^{\mathcal{A}}_\tau(a \cdot^{n+1} b, a \cdot b) \leq e^{\mathcal{A}}_\tau(a \cdot^n b, a \cdot b)$;*

  **(2)** *for all $a \in A_{\sigma\to\tau}$, $b \in A_\sigma$, for all $\epsilon > 0$ there exists $n \in \mathbf{N}$ s.t. $e^{\mathcal{A}}_\sigma(a \cdot^n b, a \cdot b) \leq \epsilon$;*

  **(3)** *(uniform non-expansiveness) $\forall n > 0 \exists \epsilon_n > 0$ s.t. $\forall \epsilon \leq \epsilon_n, \forall a, b \in A_{\sigma\to\tau}, \forall c, d \in A_\sigma$, $e^{\mathcal{A}}_{\sigma\to\tau}(a,b) \leq \epsilon$ and $e^{\mathcal{A}}_\sigma(c,d) \leq \epsilon$ implies $e^{\mathcal{A}}_\sigma(a \cdot^n c, b \cdot^n d) \leq \epsilon$.*

Conditions 1 and 2 above express the fact that the operators $\cdot^n$ approximate the behaviour of application; condition 3 replaces rule (NExp) for application.

**The Approximate $\lambda$-Algebra of the Term Model.** The $\lambda$-algebra $\mathcal{NF}$ can be extended to an approximate quantitative $\lambda$-algebra by defining operators $\cdot^n_{\sigma,\tau}$ as follows: for all $t \in NF_{\sigma\to\tau}, s \in NF_\sigma$, for all $n \in \mathbf{N}$, $t \cdot^n_{\sigma,\tau} s = t_n \cdot_{\sigma,\tau} s_n$. One can check that the approximant operators satisfy all conditions of Definition 56.

**The Approximate $\lambda$-Algebra of $D_\infty$.** The $\lambda$-algebra $\mathcal{D}$ can be extended to an approximate quantitative $\lambda$-algebra by defining operators $\cdot^n_{\sigma,\tau}$ as follows: for all $a \in D_{\sigma\to\tau}, b \in D_\sigma$, for all $n \in \mathbf{N}$, $a \cdot^n_{\sigma,\tau} b = a_{n+1} \cdot_{\sigma,\tau} b$. One can check that the approximant operators satisfy all conditions of Definition 56. Notice that the approximate algebra of $D_\infty$ yields a $\lambda$-algebra for the untyped $\lambda$-calculus.

Finally, notice that in dealing with partial and approximate algebras we have considered applicative algebras over an extended signature. For lack of space, we have not developed corresponding approximate theories including extra operators and the suitable rules on them. In particular, rule (NExp) has to be replaced by a rule expressing uniform non-expansiveness of approximant operators. We leave this as future work; here we just observe that the appropriate language for reasoning on such structures would be the *indexed $\lambda$-calculus* together with *indexed reduction*, see [5].

## 9 Conclusions

**Contributions.** This paper addresses the problem of defining quantitative algebras, in the sense of Mardare *et al.*, capable of interpreting terms of higher-order calculi. Our contributions include both negative and positive results: on the one hand we identify the main mathematical obstacles to the construction of non-trivial quantitative higher-order algebras; on the other hand we introduce quantitative variants of the traditional notions of (weak) $\lambda$-algebras, together with a sound and complete syntax, and we show that, in spite of the limitations highlighted, intriguing notions of distance for the $\lambda$-calculus do indeed exist.

**Related Work.** Since [2], metric spaces have been exploited as an alternative, quantitative, framework to standard, domain-theoretic, denotational semantics [35, 4]. The possibility of giving a metric structure to *linear* or *affine* higher-order programs is known, since Met is an SMCC, even if not a CCC. In this sense it is worth recalling the work by de Amorim *et al.* [3], along with those of Reed and Pierce [33], as well as recent work by Dahlqvist and Neves [16]. Moreover, ultra-metrics have already been used to model PCF [21]. More recently, Pistone has given a precise account of cartesian closed structure in categories of generalized metric spaces [32]. In particular, it is known that if the quantale that captures distances can vary as the types vary, as for example in the so-called differential logical relations [17], categories of generalized metric spaces can become cartesian closed. The study of metric semantics for imperative and concurrent programming languages has a long tradition [19, 18]. However, this very sophisticated apparatus is not applicable to higher order programming languages.

Partial metrics have been well-studied since [9] and [26] (where they are called $M$-sets). [10] shows that these metrics are strongly related to Scott semantics. The setting of *quantaloid-enriched* categories [25, 34] provides an abstract unifying framework for the different metric structures discussed here. In this setting, [13, 14] provide a general characterization of exponentiable morphisms and objects in categories of (generalized) metric spaces.

Finally, a somehow related approach to quantitative reasoning is provided by the use of fuzzy logic to reason about degrees of similarity between programs, as spelled out in Zadeh's pioneering work [36, 37]. More recently, fuzzy algebraic theories in the style of Mardare *et al.* have been studied [12]. However, such theories seem to lack a compositionality condition comparable to the one expressed by axiom (Nexp), hence apparently diverging from the idea of interpreting programs as non-expansive functions.

**Perspectives.** Our focus here was on the simplest case, namely that of algebras *without* a barycentric structure, thus putting ourselves in a simpler setting than the one studied by Mardare *et al.* Indeed, a natural development of this work is to study quantitative algebras for $\lambda$-calculi enriched with operations having an intrinsically quantitative content, like e.g. probabilistic choice [15] or some form of differentiation [20, 11, 28]. Another direction to explore, already suggested by some of our models, is that of exploring quantitative algebras in categories of domains like, e.g. metric CPOs [3], or continuous Scott domains (especially in virtue of their close connection with partial metric spaces [10]).

Finally, the approaches of partial and approximate algebras open new lines of investigation: suitable approximate theories are called for, and moreover the distances on programs which arise are worth to be studied in depth.

## References


1 Samson Abramsky, Esfandiar Haghverdi, and Philip J. Scott. Geometry of interaction and linear combinatory algebras. *Math. Struct. Comput. Sci.*, 12(5):625–665, 2002. doi:10.1017/S0960129502003730.

2 André Arnold and Maurice Nivat. Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *Theor. Comput. Sci.*, 11:181–205, 1980. doi:10.1016/0304-3975(80)90045-6.

3 Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. A semantic account of metric preservation. In *Proc. of POPL 2017*, pages 545–556. ACM, 2017. doi:10.1145/3009837.3009890.

4 Christel Baier and Mila E. Majster-Cederbaum. Denotational semantics in the CPO and metric approach. *Theor. Comput. Sci.*, 135(2):171–220, 1994. doi:10.1016/0304-3975(94)00046-8.

5 Henk Barendregt. *Lambda calculus, its syntax and semantics.* North-Holland, 1985. doi:10.2307/2274112.

6 Henk Barendregt, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

7 Henk Barendregt, Wil Dekkers, and Statman Richard. *Lambda calculus with Types.* Cambridge University Press, 2013. doi:10.1017/CBO9781139032636.

8 Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, 1935. doi:10.1017/S0305004100013463.

9 Michael Bukatin, Ralph Kopperman, Steve Matthews, and Homeira Pajoohesh. Partial metric spaces. *The American Mathematical Monthly*, 116(8):708–718, 2009. doi:10.4169/193009709X460831.

10 Michael A. Bukatin and Joshua S. Scott. Towards computing distances between programs via scott domains. In *Proc. of LFCS 1997*, volume 1234 of *LNCS*, pages 33–43. Springer, 1997. doi:10.1007/3-540-63045-7_4.

11 Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: incrementalizing $\lambda$-calculi by static differentiation. In *Proc. of PLDI 2014*, pages 145–155. ACM, 2014. doi:10.1145/2594291.2594304.

12 Davide Castelnovo and Marino Miculan. Fuzzy algebraic theories. In *Proc. of CSL 2022*, pages 13:1–13:17, Germany, 2022. LIPIcs. doi:10.4230/LIPIcs.CSL.2022.13.

13 Maria Manuel Clementino and Dirk Hofmann. Exponentiation in V-categories. *Topology and its Applications*, 153(16):3113–3128, 2006. doi:10.1016/j.topol.2005.01.038.

14 Maria Manuel Clementino, Dirk Hofmann, and Isar Stubbe. Exponentiable functors between quantaloid-enriched categories. *Applied Categorical Structures*, 17(1):91–101, 2009. doi:10.1201/9781498710404-10.

15 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about $\lambda$-terms: The general case. In *Proc. of ESOP 2017*, volume 10201 of *LNCS*, pages 341–367, Berlin, Heidelberg, 2017. Springer. doi:10.1007/978-3-662-54434-1_13.

16 Fredrik Dahlqvist and Renato Neves. An internal language for categories enriched over generalised metric spaces. In *Proc. of CSL 2022*, pages 13:1–13:17, Germany, 2022. LIPIcs. doi:10.4230/LIPIcs.CSL.2022.16.

17 Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. Differential logical relations, part I: the simply-typed case. In *Proc. of ICALP 2019*, volume 132 of *LIPIcs*, pages 111:1–111:14, 2019. doi:10.4230/LIPIcs.ICALP.2019.111.

18 Jaco de Bakker and Erik de Vink. *Control Flow Semantics.* MIT Press, Cambridge, MA, USA, 1996.

19 Jaco de Bakker and John-Jules Charles Meyer. Metric semantics for concurrency. In Jaco de Bakker and Jan Rutten, editors, *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, pages 104–130. World Scientific, Singapore, 1992.

20 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.

21 Martín Hötzen Escardó. A metric model of PCF. Unpublished note presented at the Workshop on Realizability Semantics and Applications, June 1999. Available at the author's webpage., 1999.

22 Raphael Espínola and Mohamed Amine Khamsi. Introduction to hyperconvex spaces. In *Handbook of Metric Fixed Point Theory*, pages 391–435. Springer Netherlands, Dordrecht, 2001. doi:10.1007/978-94-017-1748-9_13.

23 Jean-Yves Girard. Towards a geometry of interaction. *Contemporary Mathematics*, 92, 1989.

24 Joseph A. Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. doi:10.1145/947864.947865.

25 Dirk Hofmann and Isar Stubbe. Topology from enrichment: the curious case of partial metrics. *Cahiers de Topologie et Géométrie Différentielle Catégorique, LIX*, 4:307–353, 2018.

26 U. Höhle. M-valued sets and sheaves over integral commutative cl-monoids. In Stephen Ernest Rodabaugh, Erich Peter Klement, and Ulrich Höhle, editors, *Applications of Category Theory to Fuzzy Subsets*, pages 33–72. Springer Netherlands, 1992. doi:10.1007/978-94-011-2616-8_3.

27 Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography.* Chapman & Hall, 2nd edition, 2014.

28 Robert Kelly, Barak A. Pearlmutter, and Jeffrey Mark Sisking. Evolving the incremental $\lambda$-calculus into a model of forward automatic differentiation (AD). ArXiv https://arxiv.org/abs/1611.03429, 2016.

29 Radu Mardare, Prakash Panangaden, and Gordon Plotkin. Quantitative algebraic reasoning. In *Proc. of LICS 2016.* IEEE Computer Society, 2016. doi:10.1145/2933575.2934518.

30 Simone Martini. Categorical models for non-extensional $\lambda$-calculi and combinatory logic. *Math. Struct. Comput. Sci.*, 2(3):327–357, 1992. doi:10.1017/S096012950000150X.

31 Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4), 2016. doi:10.1145/2893356.

32 Paolo Pistone. On generalized metric spaces for the simply typed lambda-calculus. In *Proc. of LICS 2021.* IEEE Computer Society, 2021. doi:10.1109/LICS52264.2021.9470696.

33 Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger. *Proc. of ICFP 2010*, pages 157–168, 2010. doi:10.1145/1863543.1863568.

34 Isar Stubbe. An introduction to quantaloid-enriched categories. *Fuzzy Sets and Systems*, 256:95–116, 2014. doi:10.1016/j.fss.2013.08.009.

35 Franck van Breugel. An introduction to metric semantics: operational and denotational models for programming and specification languages. *Theor. Comput. Sci.*, 258(1):1–98, 2001. doi:10.1016/S0304-3975(00)00403-5.

36 Lofti A. Zadeh. Quantitative fuzzy semantics. *Inf. Sci.*, 3(2):159–176, 1971. doi:10.1016/S0020-0255(71)80004-X.

37 Lofti A. Zadeh. Similarity relations and fuzzy orderings. *Inf. Sci.*, 3(2):177–200, 1971. doi:10.1016/S0020-0255(71)80005-1.

# Sheaf Semantics of Termination-Insensitive Noninterference

**Jonathan Sterling**
Department of Computer Science, Aarhus University, Denmark

**Robert Harper**
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

## Abstract

We propose a new *sheaf semantics* for secure information flow over a space of abstract behaviors, based on synthetic domain theory: security classes are open/closed partitions, types are sheaves, and redaction of sensitive information corresponds to restricting a sheaf to a closed subspace. Our security-aware computational model satisfies termination-insensitive noninterference automatically, and therefore constitutes an intrinsic alternative to state of the art extrinsic/relational models of noninterference. Our semantics is the latest application of Sterling and Harper's recent re-interpretation of *phase distinctions* and noninterference in programming languages in terms of Artin gluing and topos-theoretic open/closed modalities. Prior applications include parametricity for ML modules, the proof of normalization for cubical type theory by Sterling and Angiuli, and the cost-aware logical framework of Niu *et al.* In this paper we employ the phase distinction perspective *twice*: first to reconstruct the syntax and semantics of secure information flow as a lattice of phase distinctions between "higher" and "lower" security, and second to verify the computational adequacy of our sheaf semantics with respect to a version of Abadi *et al.*'s *dependency core calculus* to which we have added a construct for declassifying termination channels.

**2012 ACM Subject Classification** Theory of computation → Abstraction; Theory of computation → Denotational semantics; Theory of computation → Categorical semantics; Theory of computation → Type theory; Security and privacy → Formal methods and theory of security

**Keywords and phrases** information flow, noninterference, denotational semantics, phase distinction, Artin gluing, modal type theory, topos theory, synthetic domain theory, synthetic Tait computability

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.5

**Related Version** *Extended Version*: `https://arxiv.org/abs/2204.09421`

**Funding** *Jonathan Sterling*: This work was supported by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.
*Robert Harper*: This research was sponsored by the United States Air Force Office of Scientific Research awards FA95502110009 and FA9550-21-1-0385 (Tristan Nguyen, program manager). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Acknowledgements** We are grateful for insightful conversations with Aslan Askarov, Stephanie Balzer, Lars Birkedal, Martín Escardó, Marcelo Fiore, Daniel Gratzer, and Tom de Jong. Thanks to Jamie Vicary for funding a visit to Cambridge during which the first author learned some of the tools needed to complete this work satisfactorily. We thank Carlos Tomé Cortiñas, Fabian Ruch, and Sandro Stucki for proof-reading.

## 1 Introduction

Security-typed languages restrict the ways that classified information can flow from high-security to low-security clients. Abadi *et al.* [1] pioneered the use of *idempotent monads* to deliver this restriction in their ***dependency core calculus*** (DCC), parameterized in a poset

© Jonathan Sterling and Robert Harper;
licensed under Creative Commons License CC-BY 4.0

of security levels $\mathcal{P}$. Covariantly in security levels $l \in \mathcal{P}$, a family of type operations $\mathsf{T}_l A$ satisfying the rules of an idempotent monad are added to the language; the idea is then that sensitive data can be hidden underneath $\mathsf{T}_l$ and unlocked only by a client with a type that can be equipped with a $\mathsf{T}_l$-algebra structure, *i.e.* a ***$\langle l \rangle$-sealed type*** in our terminology.[1] For instance, a high-security client can read a medium-security bit:

$$\begin{aligned} &\mathsf{f} : \mathsf{T_M bool} \to \mathsf{T_H bool} \\ &\mathsf{f}\, u = x \leftarrow u; \mathsf{seal_H}(\mathsf{not}\, x) \end{aligned}$$

There is however no corresponding program of type $\mathsf{T_H bool} \to \mathsf{T_M bool}$, because the type $\mathsf{T_M bool}$ of medium-security booleans is not $\langle \mathsf{H} \rangle$-sealed, *i.e.* it cannot be equipped with the structure of a $\mathsf{T_H}$-algebra. In fact, up to observational equivalence it is possible to state a ***noninterference result*** that fully characterizes such programs:

▶ **Proposition** (Noninterference). *For any closed function* $\cdot \vdash \mathsf{f} : \mathsf{T_H bool} \to \mathsf{T_M bool}$*, there exists a closed* $\cdot \vdash b : \mathsf{T_M bool}$ *such that* $\mathsf{f} \simeq \lambda_.b$*.*

Intuitively the noninterference result above follows because you cannot "escape" the monad, but to prove such a result rigorously a model construction is needed. Today the state of the art is to employ a ***relational model*** in the sense of Reynolds in which a type is interpreted as a binary relation on some domain, and a term is interpreted by a relation-preserving function. Our contribution is to introduce an ***intrinsic*** and ***non-relational semantics*** of noninterference presenting several advantages that we will argue for, inspired by the recent modal reconstruction of *phase distinctions* by Sterling and Harper [50].

## 1.1 Termination-insensitivity and the meaning of "observation"

▶ **Notation 1.** We will write $\mathsf{L}A$ for the lifting monad that Abadi *et al.* notate $A_\perp$.

When we speak of noninterference up to observational equivalence, much weight is carried by the choice of what, in fact, counts as an observation. In a functional language with general recursion, it is conventional to say that an observation is given by a computation of unit type – which necessarily either diverges or converges with the unique return value (). Under this notion of observation, noninterference up to observations takes a very strong character:

*Termination-sensitive noninterference.* For a closed partial function $\cdot \vdash \mathsf{f} : \mathsf{T_H bool} \to \mathsf{L}\,(\mathsf{T_M bool})$, either $\mathsf{f} \simeq \lambda_.\perp$ or there exists $\cdot \vdash b : \mathsf{T_M bool}$ such that $\mathsf{f} \simeq \lambda_.b$.

If on the other hand we restrict observations to only terminating computations of type bool, we evince a more relaxed ***termination-insensitive*** version of noninterference that allows leakage through the termination channel but *not* through the "return channel":

*Termination-insensitive noninterference.* For a closed partial function $\cdot \vdash \mathsf{f} : \mathsf{T_H bool} \to \mathsf{L}\,(\mathsf{T_M bool})$, given any closed $u, v$ on which $f$ terminates, we have $f u \simeq f v$.

[1] We use the term "sealing" for what Abadi *et al.* [1] call "protection"; to avoid confusion, we impose a uniform terminology to encompass both our work and that of *op. cit.* A final notational deviation on our part is that we will distinguish a security level $l \in \mathcal{P}$ from the corresponding syntactical entity $\langle l \rangle$.

## 1.2 Relational *vs.* intrinsic semantics

To verify the noninterference property for the dependency core calculus, Abadi *et al.* [1] define a ***relational semantics*** that starts from an insecure model of computation (domain theory *qua* dcpos) and restricts it by means of binary relations indexed in security levels that express the indistinguishability of sensitive bits to low-security clients. The indistinguishability relations are required to be preserved by all functions, ensuring the security properties of the model. The relational approach has an extrinsic flavor, being characterized by the *post hoc* imposition of order (noninterference) on an inherently disordered computational model. We contrast the extrinsic relational semantics of *op. cit.* with an *intrinsic* denotational semantics in which the underlying computational model has security concerns "built-in" from the start.

## 1.3 Our contribution: intrinsic semantics of noninterference

The main contribution of our paper is to develop an *intrinsic semantics* in the sense of Section 1.2, in which termination-insensitive noninterference (Section 1.1) is not bolted on but rather arises directly from the underlying computational model. To summarize our approach, instead of controlling the security properties of ordinary dcpos using a $\mathcal{P}$-indexed logical relation, we take semantics in a category of $\mathcal{P}$-indexed dcpos, *i.e.* sheaves of dcpos on a space $\mathbf{P}$ in which each security level $l \in \mathcal{P}$ corresponds to an open/closed partition. Employing the viewpoint of Sterling and Harper [50], each of these partitions induces a ***phase distinction*** between data visible below security level $l$ (open) and data that is hidden (closed), leading to a novel account of the sealing monad $\mathsf{T}_l$ as restriction to a closed subspace.

Our intrinsic semantics has several advantages over the relational approach. Firstly, termination-insensitive noninterference arises directly from our computational model. Secondly, our model of secure information flow contributes to the consolidation and unification of ideas in programming languages by treating general recursion and security typing as instances of two orthogonal and well-established notions, namely ***axiomatic & synthetic domain theory*** and ***phase distinctions*/*Artin gluing*** respectively. Termination-insensitivity then arises from the non-trivial interaction between these orthogonal layers.

In particular, our computational model is an instance of axiomatic domain theory in the sense of Fiore [10], and embeds into a sheaf model of synthetic domain theory [14, 9, 12, 13, 11, 15, 30]. Hence the interpretation of the PCF fragment of DCC is interpreted exactly as in the standard Plotkin semantics of general recursion in categories of partial maps, in contrast to the relational model of Abadi *et al.* Lastly, the view of security levels as phase distinctions per Sterling and Harper [50] advances a uniform perspective on noninterference scenarios that has already proved fruitful for resolving several problems in programming languages:

1. A generalized abstraction theorem for ML modules with strong sums [50].
2. Normalization and decidability of type checking for cubical type theory [49, 48] and multi-modal type theory [17]; guarded canonicity for guarded dependent type theory [18].
3. The design and metatheory of the **calf** logical framework [31] for simultaneously verifying the correctness and complexity of functional programs.

The final benefit of the phase distinction perspective is that logical relations arguments can be re-cast as imposing an *additional* orthogonal phase distinction between *syntax* and *logic/specification*, an insight originally due to Peter Freyd in his analysis of the existence and disjunction properties in terms of Artin gluing [16]. We employ this insight in the present paper to develop a uniform treatment of our denotational semantics and its computational adequacy in terms of phase distinctions.

## 2 Background: relational semantics of noninterference

To establish noninterference for the dependency core calculus, Abadi *et al.* [1] define a relational model of their monadic language in which each type $A$ is interpreted as a dcpo $|A|$ equipped with a family of admissible binary relations $R_l^A$ indexed in security levels $l \in \mathcal{P}$. In the relational semantics, a term $\Gamma \vdash M : A$ is interpreted as a continuous function $|M| : |\Gamma| \longrightarrow |A|$ such that for all $l \in \mathcal{P}$, if $\gamma\ R_l^\Gamma\ \gamma'$ then $|M|\gamma\ R_l^A\ |M|\gamma'$.

▶ Remark 2. Two elements $u, v \in A$ such that $u\ R_l^A\ v$ have been called *equivalent* in subsequent literature, but this terminology may lead to confusion as there is nothing forcing the relation to be transitive, nor even symmetric nor reflexive.

The essence of the relational model is to impose *relations* between elements that should not be distinguishable by a certain security class; a type like bool or string whose relation is totally discrete, then, allows any security class to distinguish all distinct elements. Non-discrete types enter the picture through the sealing modality $\mathsf{T}_l$:

$$|\mathsf{T}_l A| = |A| \qquad u\ R_k^{\mathsf{T}_l A}\ v \iff \begin{cases} u\ R_k^A\ v & \text{if } l \sqsubseteq k \\ \top & \text{otherwise} \end{cases}$$

Under this interpretation, the denotation of a function $\mathsf{T}_\mathsf{H}\mathsf{bool} \to \mathsf{T}_\mathsf{M}\mathsf{bool}$ must be a constant function, as $u\ R_\mathsf{H}^{\mathsf{bool}}\ v$ if and only iff $u = v$. By proving computational adequacy for this denotational semantics, one obtains the analogous *syntactic* noninterference result up to observational equivalence.

*Generalization and representation of relational semantics.* The relations imposed on each type give rise to a form of cohesion in the sense of Lawvere [28], where elements that are related are thought of as "stuck together". Then noninterference arises from the behavior of maps from a relatively codiscrete space into a relatively discrete space, as pointed out by Kavvos [25] in his *tour de force* generalization of the relational account of noninterference in terms of axiomatic cohesion. Another way to understand the relational account is by *representation*, as attempted by Tse and Zdancewic [54] and executed by Bowman and Ahmed [6]: one may embed DCC into a polymorphic lambda calculus in which the security abstraction is implemented by *actual* type abstraction.

### Adapting the relational semantics for termination-insensitivity

In the relational semantics of the dependency core calculus, the termination-sensitive version of noninterference is achieved by interpreting the *lift* of a type in the following way:

$$|A_\perp| = |A|_\perp \qquad u\ R_l^{A_\perp}\ v \iff (u, v\downarrow \wedge\ u\ R_l^A\ v) \vee (u = v = \perp)$$

To adapt the relational semantics for termination-insensitivity, Abadi *et al.* change the interpretation of lifts to identify *all* elements with the bottom element:

$$|A_\perp| = |A|_\perp \qquad u\ R_l^{A_\perp}\ v \iff (u, v\downarrow \wedge\ u\ R_l^A\ v) \vee (u = \perp) \vee (v = \perp)$$

That all data is "indistinguishable" from the non-terminating computation means that the indistinguishability relation cannot be both transitive and non-trivial, a somewhat surprising state of affairs that leads to our critique of relational semantics for information flow below and motivates our new perspective based on the analogy between *phase distinctions* in programming languages and *open/closed partitions* in topological spaces [50].

### Critique of relational semantics for information flow

From our perspective there are several problems with the relational semantics of Abadi *et al.* [1] that, while not fatal on their own, inspire us to search for an alternative perspective.

*Failure of monotonicity.* First of all, within the context of the relational semantics it would be appropriate to say that an object $A$ is $\langle l\rangle$-sealed when $A \longrightarrow \mathsf{T}_l A$ is an isomorphism. But in the semantics of Abadi *et al.*, it is not necessarily the case that a $\langle l\rangle$-sealed object is $\langle k\rangle$-sealed when $k \sqsubseteq l$. It is true that objects that are *definable* in the dependency core calculus are better behaved, but in proper denotational semantics one is not concerned with the image of an interpretation function but rather with the entire category.

*Failure of transitivity.* A more significant and harder to resolve problem is the fact that the indistinguishability relation $R^A_l$ assigned to each type cannot be construed as an equivalence relation – despite the fact that in real life, indistinguishability is indeed reflexive, symmetric, and transitive. As we have pointed out, the adaptation of DCC's relational semantics for termination-insensitivity is evidently incompatible with using (total or partial) equivalence relations to model indistinguishability, as transitivity would ensure that no two elements of $A_\perp$ can be distinguished from another.

*Where is the dominance?* Conventionally the denotational semantics for a language with general recursion begins by choosing a category of "predomains" and then identifying a notion of *partial map* between them that evinces a *dominance* [10, 42]. It is unclear in what sense the DCC's relational semantics reflects this hard-won arrangement; as we have seen, the adaptation of the relational semantics for termination-insensitivity further increases the distance from ordinary domain-theoretic semantics.

**Perspective.** Abadi *et al.*'s relational semantics is based on imposing secure information flow properties on an existing insecure model of partial computation, but this is quite distinct from an *intrinsic denotational semantics* for secure information flow – which would necessarily entail new notions of predomain and partial map that are sensitive to security from the start. In this paper we report on such an intrinsic semantics for secure information flow in which termination-insensitive noninterference arises inexorably from the chosen dominance.

## 3 Central ideas of this paper

In this section, we dive a little deeper into several of the main concepts that substantiate the contributions of this paper. We begin by fixing a poset $\mathcal{P}$ of security levels closed under finite meets, for example $\mathcal{P} = \{\mathsf{L} \sqsubset \mathsf{M} \sqsubset \mathsf{H} \sqsubset \top\}$. The purpose of including a security level even higher than $\mathsf{H}$ will become apparent when we explain the meaning of the sealing monad $\mathsf{T}_l$.

▶ **Notation 3.** Given a space $\mathbf{X}$ and an open set $U \in \mathcal{O}_{\mathbf{X}}$, we will write $\mathbf{X}_{/U}$ for the open subspace spanned by $U$ and $\mathbf{X}_{\bullet U}$ for the corresponding complementary closed subspace. We also will write $\mathcal{S}_{\mathbf{X}}$ for the category of sheaves on the space $\mathbf{X}$.

### 3.1 A space of abstract behaviors and security policies

We begin by transforming the security poset $\mathcal{P}$ into a topological space $\mathbf{P}$ of "abstract behaviors" whose algebra of open sets $\mathcal{O}_{\mathbf{P}}$ can be thought of as a lattice of *security policies* that govern whether a given behavior is permitted.

▶ **Definition 4.** *An **abstract behavior** is a* filter *on the poset $\mathcal{P}$,* i.e. *a monotone subset $x \subseteq \mathcal{P}$ such that $\bigwedge_{i<n} l_i \in x$ if and only if each $l_i \in x$.*

▶ **Definition 5.** *A* ***security policy*** *is a lower set in* $\mathcal{P}$*,* i.e. *an antitone subset* $U \subseteq \mathcal{P}$*. We will write* $U \Vdash x$ *to mean* $U$ *permits the behavior* $x$*,* i.e. *the subset* $x \cap U$ *is inhabited.*

An abstract behavior $x$ denotes the set of security levels $l \in \mathcal{P}$ at which it is permitted; a security policy $U$ denotes the set of security levels *above which* some behavior is permitted.

▶ **Construction 6.** We define $\mathbf{P}$ to be the topological space whose points are abstract behaviors, and whose open sets are of the form $\{x \mid U \Vdash x\}$ for some security policy $U$.[2]

We have a meet-preserving embedding of posets $\langle - \rangle : \mathcal{P} \hookrightarrow \mathcal{O}_{\mathbf{P}}$ that exhibits $\mathcal{O}_{\mathbf{P}}$ as the free completion of $\mathcal{P}$ under joins, or equivalently the free frame on the meet semi-lattice $\mathcal{P}$.

▶ **Intuition 7** (Open and closed subspaces)**.** Each security level $l \in \mathcal{P}$ represents a security policy $\langle l \rangle \in \mathcal{O}_{\mathbf{P}}$ whose corresponding open subspace $\mathbf{P}_{/\langle l \rangle}$ is spanned by the behaviors *permitted* at security levels $l$ and above. Conversely the complementary closed subspace $\mathbf{P}_{\bullet\langle l \rangle} = \mathbf{P} \setminus \mathbf{P}_{/\langle l \rangle}$ is spanned by behaviors that are *forbidden* at security level $l$ and below.

## 3.2 Sheaves on the space of abstract behaviors

Our intention is to interpret each type of a dependency core calculus as a *sheaf* on the space $\mathbf{P}$ of abstract behaviors. To see why this interpretation is plausible as a basis for secure information flow, we note that a sheaf on $\mathbf{P}$ is the same thing as a presheaf on the poset $\mathcal{P}$, *i.e.* a family of sets $(A_l)_{l \in \mathcal{P}}$ indexed contravariantly in $\mathcal{P}$ in the sense that for $k \sqsubseteq l$ there is a chosen restriction function $A_l \to A_k$ satisfying two laws. Hence a sheaf on $\mathbf{P}$ determines (1) for each security level $l \in \mathcal{P}$ a choice of what data is visible under the security policy $\langle l \rangle$, and (2) a way to *redact* data as it passes under a more restrictive security policy $\langle k \rangle \subseteq \langle l \rangle$.

## 3.3 Transparency and sealing from open and closed subspaces

For any subspace $\mathbf{Q} \subseteq \mathbf{P}$, a sheaf $A \in \boldsymbol{\mathcal{S}}_{\mathbf{P}}$ can be restricted to $\mathbf{Q}$, and then extended again to $\mathbf{P}$. This composite operation gives rise to an *idempotent monad* on $\boldsymbol{\mathcal{S}}_{\mathbf{P}}$ that has the effect of purging any data from $A \in \boldsymbol{\mathcal{S}}_{\mathbf{P}}$ that cannot be seen from the perspective of $\mathbf{Q}$. The idempotent monads corresponding to the open and closed subspaces induced by a security level $l \in \mathcal{P}$ are named and notated as follows:

1. The ***transparency monad*** $A \mapsto (\langle l \rangle \Rightarrow A)$ replaces $A$ with whatever part of it can be viewed under the policy $\langle l \rangle$. The transparency monad is the function space $A^{\langle l \rangle}$, recalling that an open set of $\mathbf{P}$ is the same as a subterminal sheaf. When the unit is an isomorphism at $A$, we say that $A$ is $\langle l \rangle$***-transparent***.
2. The ***sealing monad*** $A \mapsto (\langle l \rangle \bullet A)$ removes from $A$ whatever part of it can be viewed under the policy $\langle l \rangle$. The sealing monad can be constructed as the pushout $\langle l \rangle \sqcup_{\langle l \rangle \times A} A$. When the unit is an isomorphism at $A$, we say that $A$ is $\langle l \rangle$***-sealed***.

The transparency and sealing monads interact in two special ways, which can be made apparent by appealing to the visualization of their behavior that we present in Figure 1.

1. The $\langle l \rangle$-transparent part of a $\langle l \rangle$-sealed sheaf is trivial, *i.e.* we have $(\langle l \rangle \Rightarrow (\langle l \rangle \bullet A)) \cong \{\bigstar\}$.
2. Any sheaf $A \in \boldsymbol{\mathcal{S}}_{\mathbf{P}}$ can be reconstructed as the fiber product $(\langle l \rangle \Rightarrow A) \times_{\langle l \rangle \bullet (\langle l \rangle \Rightarrow A)} \langle l \rangle \bullet A$.

---

[2] Those familiar with the point-free topology of topoi [23, 55, 2] will recognize that $\mathbf{P}$ is more simply described as the presheaf topos $\widehat{\mathcal{P}}$: viewed as a space, it is the dcpo completion of $\mathcal{P}^{\mathsf{op}}$, and as a frame it is the free cocompletion of $\mathcal{P}$. The definition of $U \Vdash x$ then presents a computation of the *stalk* $U_x$ of the subterminal sheaf $U \in \boldsymbol{\mathcal{S}}_{\mathbf{P}}$ at the behavior $x \in \mathbf{P}$.

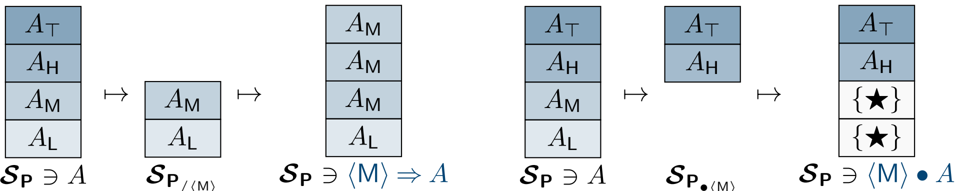



**Figure 1** The transparency and sealing monads for $\mathsf{M} \in \mathcal{P}$ on a sheaf $A \in \mathcal{S}_\mathsf{P}$ visualized.

The first property above immediately gives rise to a form of noninterference, which justifies our intent to interpret DCC's sealing monad as $\mathsf{T}_l A = \langle l \rangle \bullet A$.

▶ **Observation 8** (Noninterference)**.** *Any map* $\langle l \rangle \bullet A \longrightarrow \mathsf{bool}$ *is constant.*

**Proof.** We may verify that the boolean sheaf $\mathsf{bool}$ is $\langle l \rangle$-transparent for all $l \in \mathcal{P}$. ◀

Our sealing monad above is well-known to the type-and-topos–theoretic community as the ***closed modality*** [41, 43, 3] corresponding to the open set $\langle l \rangle \in \mathcal{O}_\mathsf{P}$. In the context of (total) dependent type theory, our sealing monad has excellent properties not shared by those of Abadi *et al.* [1], such as justifying *dependent* elimination rules and commuting with identity types. In contrast to the *classified sets* of Kavvos [25] which cannot form a topos, our account of information flow is compatible with the full internal language of a topos.

## 3.4 Recursion and termination-insensitivity via sheaves of domains

To incorporate recursion into our sheaf semantics of information flow, in this section we consider *internal dcpos* in $\mathcal{S}_\mathsf{P}$, *i.e.* sheaves of dcpos. Later in the technical development of our paper, we work in the axiomatic setting of synthetic domain theory, but all the necessary intuitions can also be understood concretely in terms of dcpos. Domain theory internal to $\mathcal{S}_\mathsf{P}$ works very similarly to classical domain theory, but it must be developed without appealing to the law of the excluded middle or the axiom of choice as these do not hold in $\mathcal{S}_\mathsf{P}$ except for a particularly degenerate security poset. De Jong and Escardó [8] explain how to set up the basics of domain theory in a suitably constructive manner, which we will not review.

The sheaf-theoretic domain semantics sketched above leads immediately to a new and simplified account of termination-insensitivity. It is instructive to consider whether there is an analogue to Observation 8 for partial continuous functions $\langle l \rangle \bullet A \longrightarrow \mathsf{L}\,\mathsf{bool}$. It is not the case that $\mathsf{L}\,\mathsf{bool}$ is $\langle l \rangle$-transparent for all $l \in \mathcal{P}$, so it would not follow that any continuous map $\langle l \rangle \bullet A \longrightarrow \mathsf{L}\,\mathsf{bool}$ is constant. A partial function always extends to a total function on a restricted domain, however, so we may immediately conclude the following:

▶ **Observation 9** (Termination-insensitive noninterference)**.** *For any continuous map* $f : \langle l \rangle \bullet A \longrightarrow \mathsf{L}\,\mathsf{bool}$ *and elements* $u, v : \langle l \rangle \bullet A$ *with* $fu$ *and* $fv$ *defined, we have* $fu = fv$.

This is the sense in which termination-insensitive noninterference arises automatically from the combination of domain theory with sheaf semantics for information flow.

# 4 Refined dependency core calculus

We now embark on the technical development of this paper, beginning with a call-by-push-value (cbpv) style [29] refinement of the dependency core calculus over a poset $\mathcal{P}$ of security levels. We will work informally in the logical framework of locally Cartesian closed categories *à la* Gratzer and Sterling [20]; we will write $\mathfrak{T}$ for the free locally Cartesian closed category generated by all the constants and equations specified herein.

## 4.1 The basic language

We have value types $A : \mathsf{tp}^{+}$ and computation types $X : \mathsf{tp}^{\ominus}$; because our presentation of cbpv does not include stacks, we will not include a separate syntactic category for computations but instead access them through thunking. The sorts of value and computation types and their adjoint connectives are specified below:

$$\mathsf{tp}^{+}, \mathsf{tp}^{\ominus} : \mathbf{Sort} \qquad \mathsf{tm} : \mathsf{tp}^{+} \to \mathbf{Sort} \qquad \mathsf{U} : \mathsf{tp}^{\ominus} \to \mathsf{tp}^{+} \qquad \mathsf{F} : \mathsf{tp}^{+} \to \mathsf{tp}^{\ominus}$$

We let $A, B, C$ range over $\mathsf{tp}^{+}$ and $X, Y, Z$ over $\mathsf{tp}^{\ominus}$. We will often write $A$ instead of $\mathsf{tm}\,A$ when it causes no ambiguity. Free computation types are specified as follows:

$$\begin{array}{ll} \mathsf{ret} : A \to \mathsf{UF}A & \mathsf{bind}\,(\mathsf{ret}\,u)\,f \equiv_{\mathsf{U}X} f\,u \\ \mathsf{bind} : \mathsf{UF}A \to (A \to \mathsf{U}X) \to \mathsf{U}X & \mathsf{bind}\,u\,\mathsf{ret} \equiv_{\mathsf{UF}A} u \\ & \mathsf{bind}\,(\mathsf{bind}\,u\,f)\,g \equiv_{\mathsf{U}X} \mathsf{bind}\,u\,(\lambda x.\mathsf{bind}\,(f\,x)\,g) \end{array}$$

We support general recursion in computation types:

$$\mathsf{fix} : (\mathsf{U}X \to \mathsf{U}X) \to \mathsf{U}X \qquad \mathsf{fix}\,f \equiv f\,(\mathsf{fix}\,f)$$

We close the universe $X : \mathsf{tp}^{\ominus} \vdash \mathsf{tm}\,\mathsf{U}X$ of computation types and thunked computations under all function types $\mathsf{tm}\,A \to \mathsf{tm}\,\mathsf{U}X$ by adding a new computation type constant $\mathsf{fn}$ equipped with a universal property like so:

$$\mathsf{fn} : \mathsf{tp}^{+} \to \mathsf{tp}^{\ominus} \to \mathsf{tp}^{\ominus} \qquad \mathsf{fn.tm} : (A \to \mathsf{U}X) \cong \mathsf{U}\,(\mathsf{fn}\,A\,X)$$

We will treat this isomorphism implicitly in our informal notation, writing $\lambda x.u(x)$ for both meta-level and object-level function terms. Finite product types are specified likewise:

$$\begin{array}{ll} \mathsf{prod} : \mathsf{tp}^{+} \to \mathsf{tp}^{+} \to \mathsf{tp}^{+} & \mathsf{unit} : \mathsf{tp}^{+} \\ \mathsf{prod.tm} : A \times B \cong \mathsf{prod}\,A\,B & \mathsf{unit.tm} : \mathbf{1} \cong \mathsf{unit} \end{array}$$

Sum types must be treated specially because we do not intend them to be coproducts in the logical framework: they should have a universal property for types, not for sorts.

$$\begin{array}{ll} \mathsf{sum} : \mathsf{tp}^{+} \to \mathsf{tp}^{+} \to \mathsf{tp}^{+} & \mathsf{case} : \mathsf{sum}\,A\,B \to (A \to C) \to (B \to C) \to C \\ \mathsf{inl} : A \to \mathsf{sum}\,A\,B & \mathsf{case}\,(\mathsf{inl}\,u)\,f\,g \equiv_C f\,u \\ \mathsf{inr} : B \to \mathsf{sum}\,A\,B & \mathsf{case}\,(\mathsf{inr}\,v)\,f\,g \equiv_C g\,v \\ & \mathsf{case}\,u\,(\lambda x.f\,(\mathsf{inl}\,x))\,(\lambda x.f\,(\mathsf{inr}\,x)) \equiv_C f\,u \end{array}$$

## 4.2 The sealing modality and declassification

For each $l \in \mathcal{P}$, we add an *abstract* proof irrelevant proposition $\langle l \rangle : \mathbf{Prop}$ to the language; this proposition represents the condition that the "client" has a lower security clearance than $l$. This "redaction" is implemented by isolating the types that are *sealed* at $\langle l \rangle$, *i.e.* those that become singletons in the presence of $\langle l \rangle$:

$$\begin{array}{lcl} \langle l \rangle : \mathbf{Prop} & & \mathsf{sealed}_{\langle l \rangle} : \mathsf{tp}^+ \to \mathbf{Prop} \\ \langle k \rangle \to \langle l \rangle & (k \leq l \in \mathcal{P}) & \mathsf{sealed}_{\langle l \rangle}\, A :\equiv \langle l \rangle \to \{x : A \mid \forall y : A. x \equiv_A y\} \\ \langle k \rangle \to \langle l \rangle \to \langle k \wedge l \rangle & & \end{array}$$

We will write $\mathsf{tp}^+_{\bullet\langle l \rangle} \subseteq \mathsf{tp}^+$ for the subtype spanned by value types $A$ for which $\mathsf{sealed}_{\langle l \rangle}\, A$ holds. As in Section 3.3, we will write $\bigstar$ for the unique element of an $\langle l \rangle$-sealed type in the presence of $u : \langle l \rangle$. Next we add the sealing modality itself:

$$\begin{array}{ll} \mathsf{T}_l : \mathsf{tp}^+ \to \mathsf{tp}^+_{\bullet\langle l \rangle} & \mathsf{unseal}_l : \left\{B : \mathsf{tp}^+_{\bullet\langle l \rangle}\right\} \to \mathsf{T}_l\, A \to (A \to B) \to B \\ \mathsf{seal}_l : A \to \mathsf{T}_l\, A & \mathsf{unseal}_l\, (\mathsf{seal}_l\, u)\, f \equiv_B f\, u \\ & \mathsf{unseal}_l\, u\, (\lambda x. f\, (\mathsf{seal}_l\, x)) \equiv_B f\, u \end{array}$$

Finally a construct for declassifying the termination channel of a sealed computation:

$$\mathsf{tdcl}_{\langle l \rangle} : \left\{A : \mathsf{tp}^+_{\bullet\langle l \rangle}\right\} \to \mathsf{T}_l \mathsf{UF} A \to \mathsf{UF} A \qquad \mathsf{tdcl}_{\langle l \rangle}\, (\mathsf{seal}_l\, (\mathsf{ret}\, u)) \equiv_{\mathsf{UF}A} \mathsf{ret}\, u$$

▶ Remark 10. The $\langle l \rangle$ propositions play a purely book-keeping role, facilitating verification of program equivalences in the same sense as the ghost variables of Owicki and Gries [33].

# 5 Denotational semantics in synthetic domain theory

We will define our denotational semantics for information flow and termination-insensitive noninterference in a category of domains indexed in $\mathcal{P}$. To give a model of the theory presented in Section 4 means to define a locally Cartesian closed functor $\mathcal{T} \longrightarrow \mathcal{E}$ where $\mathcal{E}$ is locally Cartesian closed. Unfortunately no category of domains can be locally Cartesian closed, but we can *embed* categories of domains in a locally Cartesian closed category by following the methodology of ***synthetic domain theory*** [14, 9, 12, 13, 11, 15, 30].[3]

## 5.1 A topos for information flow logic

Recall that $\mathcal{P}$ is a poset of security levels closed under finite meets. The presheaf topos $\mathbf{P}$ defined by the identification $\boldsymbol{\mathcal{S}}_{\mathbf{P}} = [\mathcal{P}^{\mathsf{op}}, \mathsf{Set}]$ contains propositions $\mathsf{y}_{\mathcal{P}} l$ corresponding to every security level $l \in \mathcal{P}$, and is closed under both sealing and transparency modalities $\mathsf{y}_{\mathcal{P}} l \Rightarrow E, \mathsf{y}_{\mathcal{P}} l \bullet E$ in the sense of Section 3.3; in more traditional parlance, these are the *open* and *closed* modalities corresponding to the proposition $\mathsf{y}_{\mathcal{P}} l$ [41]. It is possible to give a denotational semantics for a *total* fragment of our language in $\boldsymbol{\mathcal{S}}_{\mathbf{P}}$, but to interpret recursion we need some kind of domain theory. We therefore define a topos model of synthetic domain theory that lies over $\mathbf{P}$ and hence incorporates the information flow modalities seamlessly.

[3] In particular we focus on the style of synthetic domain theory based on Grothendieck topoi and well-complete objects. There is another very productive strain of synthetic domain theory based on realizability and replete objects that has different properties [22, 34, 53, 35, 36, 37, 38, 39].

## 5.2 Synthetic domain theory over the information flow topos

We will now work abstractly with a Grothendieck topos $\mathbf{C}$ equipped with a dominance $\Sigma \in \mathcal{S}_{\mathbf{C}}$, called the ***Sierpiński space***, satisfying several axioms that give rise to a reflective subcategory of objects that behave like predomains. We leave the construction of $\mathbf{C}$ to our extended version, where it is built by adapting the recipe of Fiore and Plotkin [12].

▶ **Definition 11** (Rosolini [42]). *A* ***dominion*** *on a category $\mathcal{E}$ is a stable class of monos closed under identity and composition. Given a dominion $\mathcal{M}$ such that $\mathcal{E}$ has finite limits, a* ***dominance*** *for $\mathcal{M}$ is a classifier $\top : \mathbf{1}_{\mathcal{E}} \rightarrowtail \Sigma$ for the elements of $\mathcal{M}$ in the sense that every $U \rightarrowtail A \in \mathcal{M}$ gives rise to a* unique *map $\chi_U : A \longrightarrow \Sigma$ such that $U \cong \chi_U^* \top$.*

If $\mathcal{E}$ is locally cartesian closed, we may form the ***partial element classifier*** monad $\mathsf{L} : \mathcal{E} \longrightarrow \mathcal{E}$ for a dominance $\Sigma$, setting $\mathsf{L}E = \sum_{\phi:\Sigma} \phi \Rightarrow E$; given $e \in \mathsf{L}E$, we will write $e{\downarrow} \in \Sigma$ for the termination support $\pi_1\, e$ of $e$. We are particularly interested in the case where $\mathsf{L}$ has a final coalgebra $\bar{\boldsymbol{\omega}} \cong \mathsf{L}\bar{\boldsymbol{\omega}}$ and an initial algebra $\mathsf{L}\boldsymbol{\omega} \cong \boldsymbol{\omega}$. When $\mathcal{E}$ is the category of sets, $\boldsymbol{\omega}$ is just the natural numbers object $\mathbb{N}$ and $\bar{\boldsymbol{\omega}}$ is $\mathbb{N}_\infty$, the natural numbers with an infinite point adjoined. In general, one should think of $\boldsymbol{\omega}$ as the "figure shape" of a formal $\omega$-chain $\boldsymbol{\omega} \longrightarrow E$ that takes into account the data of the dominance; then $\bar{\boldsymbol{\omega}}$ is the figure shape of a formal $\omega$-chain equipped with its supremum, given by evaluation at the infinite point $\infty \in \bar{\boldsymbol{\omega}}$. There is a canonical inclusion $\boldsymbol{\iota} : \boldsymbol{\omega} \rightarrowtail \bar{\boldsymbol{\omega}}$ witnessing the *incidence relation* between a chain equipped with its supremum and the underlying chain.

▶ **Axiom SDT-1.** *$\Sigma$ has* ***finite joins*** *$\bigvee_{i<n} \phi_i$ that are preserved by the inclusion $\Sigma \subseteq \Omega$. We will write $\bot$ for the empty join and $\phi \vee \psi$ for binary joins.*

▶ **Definition 12** (Complete types). *In the internal language of $\mathcal{E}$, a type $E$ is called* ***complete*** *when it is internally orthogonal to the comparison map $\boldsymbol{\omega} \rightarrowtail \bar{\boldsymbol{\omega}}$. In the internal language, this says that for any formal chain $e : \boldsymbol{\omega} \longrightarrow E$ there exists a* unique *figure $\hat{e} : \bar{\boldsymbol{\omega}} \longrightarrow E$ such that $\hat{e} \circ \boldsymbol{\iota} = e$. In this scenario, we write $\bigsqcup_{i \in \boldsymbol{\omega}} e\, i$ for the evaluation $\hat{e}\, \infty$.*

▶ **Axiom SDT-2.** *The initial lift algebra $\boldsymbol{\omega}$ is the colimit of the following $\omega$-chain of maps:*

$$\emptyset \xrightarrow{\quad ! \quad} \mathsf{L}\emptyset \xrightarrow{\quad \mathsf{L}! \quad} \mathsf{L}^2\emptyset \xrightarrow{\quad \mathsf{L}^2! \quad} \ldots$$

▶ **Definition 13.** *A type $E$ is called a* ***predomain*** *when $\mathsf{L}E$ is complete.*

▶ **Axiom SDT-3.** *The dominance $\Sigma$ is a predomain.*

The category of predomains is complete, cocomplete, closed under lifting, exponentials, and powerdomains, and is a reflective exponential ideal in $\mathcal{S}_{\mathbf{C}}$ – thus better behaved than any classical category of predomains. The predomains with $\mathsf{L}$-algebra structure serve as an appropriate notion of *domain* in which arbitrary fixed points can be interpreted by taking the supremum of formal $\boldsymbol{\omega}$-chains of approximations $f^n \bot$; in addition to "term-level" recursion, we may also interpret recursive types. We impose two additional axioms for information flow:

▶ **Axiom SDT-4.** *The topos $\mathbf{C}$ is equipped with a geometric morphism $\boldsymbol{p}_{\mathbf{C}} : \mathbf{C} \longrightarrow \mathbf{P}$ such that the induced functor $\boldsymbol{p}_{\mathbf{C}}^* \mathsf{y}_{\mathcal{P}} : \mathcal{P} \longrightarrow \mathcal{O}_{\mathbf{C}}$ is fully faithful and is valued in $\Sigma$-propositions. We will write $\langle l \rangle$ for each $\boldsymbol{p}_{\mathbf{C}}^* \mathsf{y}_{\mathcal{P}} l$.*

Axiom SDT-4 ensures that our domain theory include computations whose termination behavior depends on the observer's security level. The following Axiom SDT-5 is applied to the semantic noninterference property.

▶ **Axiom SDT-5.** *Any constant object* $\mathsf{C}^*[n] \in \boldsymbol{\mathcal{S}}_{\mathsf{C}}$ *for* $[n]$ *a finite set is an* $\langle l \rangle$*-transparent predomain for any* $l \in \mathfrak{P}$.

The category $\boldsymbol{\mathcal{S}}_{\mathsf{C}}$ is closed under as many topos-theoretic universes [51] as there are Grothendieck universes in the ambient set theory. For any such universe $\mathbf{U}_i$, there is a subuniverse $\mathbf{Predom}_i \subseteq \mathbf{U}_i$ spanned by *predomains*; we note that being a predomain is a property and not a structure. The object $\mathbf{Predom}_i$ can exist because being a predomain is a local property that can be expressed in the internal logic. In fact, the predomains can be seen to be not only a reflective subcategory but also a reflective *subfibration* as they are obtained by the internal localization at a class of maps [45]; therefore the reflection can be internalized as a connective $\mathbf{U}_i \longrightarrow \mathbf{Predom}_i$ implemented as a quotient-inductive type [44]. We may define the corresponding universe of domains $\mathbf{Dom}_i$ to be the collection of predomains in $\mathbf{Predom}_i$ equipped with $\mathsf{L}$-algebra structures. We hereafter suppress universe levels.

## 5.3 The stabilizer of a predomain and its action

In this section, we work internally to the synthetic domain theory of $\boldsymbol{\mathcal{S}}_{\mathsf{C}}$; first we recall the definition of an *action* for a commutative monoid.

▶ **Definition 14.** *Let* $(M, 0, +)$ *be a monoid object in the category of predomains; an* ***$M$-action structure*** *on a predomain* $A$ *is given by a function* $\|_A : M \times A \longrightarrow A$ *satisfying the identities* $0 \|_A a = a$ *and* $m \|_A n \|_A a = (m + n) \|_A a$.

Write $\Sigma^{\vee}$ for the additive monoid structure of the Sierpiński domain, with addition given by $\Sigma$-join $\phi \vee \psi$ and the unit given by the non-terminating computation $\bot$. Our terminology below is inspired by stabilizer subgroups in algebra.

▶ **Definition 15** (The stabilizer of a predomain). *Given a predomain* $A$*, we define the* ***stabilizer*** *of* $A$ *to be the submonoid* $\mathsf{Stab}_{\Sigma^{\vee}} A \subseteq \Sigma^{\vee}$ *spanned by* $\phi : \Sigma^{\vee}$ *such that* $A$ *is* $\phi$*-sealed,* i.e. *the projection map* $A \times \phi \longrightarrow \phi$ *is an isomorphism.*

▶ Remark 16. We can substantiate the analogy between Definition 15 and stabilizer subgroups in algebra. Up to coherence issues that could be solved using higher categories, any category $\mathfrak{P}$ of predomains closed under subterminals and pushouts can be structured with a monoid action over $\Sigma^{\vee}$; the action $\|_{\mathfrak{P}} : \Sigma^{\vee} \times \mathfrak{P} \longrightarrow \mathfrak{P}$ takes $A$ to the $\phi$-sealed object $\phi \|_{\mathfrak{P}} A \coloneqq \phi \bullet A$. Up to isomorphism, the identities for a $\Sigma^{\vee}$-action can be seen to be satisfied. Then we say that the stabilizer of a predomain $A \in \mathfrak{P}$ is the submonoid $\mathsf{Stab}_{\Sigma^{\vee}} A \subseteq \Sigma^{\vee}$ consisting of propositions $\phi$ such that $\phi \|_{\mathfrak{P}} A \cong A$.

▶ **Lemma 17.** *For any predomain* $A$*, we may define a canonical* $\mathsf{Stab}_{\Sigma^{\vee}} A$*-action on* $\mathsf{L}A$*:*

$$\|_{\mathsf{L}A} : \mathsf{Stab}_{\Sigma^{\vee}} A \times \mathsf{L}A \to \mathsf{L}A$$
$$\phi \|_{\mathsf{L}A}\, a = (\phi \vee a{\downarrow}, [\phi \hookrightarrow \bigstar, a{\downarrow} \hookrightarrow a])$$

The stabilizer action described in Lemma 17 will be used to implement declassification of termination channels in our denotational semantics.

▶ **Lemma 18.** *The stabilizer action preserves terminating computations in the sense that* $\phi \|_{\mathsf{L}A}\, u = u$ *for* $\phi : \mathsf{Stab}_{\Sigma^{\vee}} A$ *and terminating* $u : \mathsf{L}A$.

**Proof.** We observe that $\phi \vee \top = \top$, hence for terminating $a$ we have $\phi \|_{\mathsf{L}A}\, a = a$. ◀

## 5.4 The denotational semantics

We now define an algebra for the theory $\mathfrak{T}$ in $\boldsymbol{\mathcal{S}}_{\mathsf{C}}$; the initial prefix of this algebra is standard:

$$\begin{array}{ll}
[\![\mathsf{tp}^{+}]\!] = \mathbf{Predom} & [\![\mathsf{prod}]\!]\,A\,B = A \times B \\
[\![\mathsf{tp}^{\ominus}]\!] = \mathbf{Dom} & [\![\mathsf{prod.tm}]\!] = \langle\text{canonical}\rangle \\
[\![\mathsf{U}]\!]\,X = X & [\![\mathsf{unit}]\!] = \mathbf{1}_{\mathbf{Predom}} \\
[\![\mathsf{F}]\!]\,A = \mathsf{L}A & [\![\mathsf{unit.tm}]\!] = \langle\text{canonical}\rangle \\
[\![\mathsf{ret}]\!]\,a = a & [\![\mathsf{sum}]\!]\,A\,B = A + B \\
[\![\mathsf{bind}]\!]\,m\,f = f^{\sharp}\,m & [\![\mathsf{inl}]\!]\,a = \mathsf{inl}\,a \\
[\![\mathsf{fix}]\!]\,f = \mathsf{fix}\,f & [\![\mathsf{inr}]\!]\,a = \mathsf{inr}\,a \\
[\![\mathsf{fn}]\!]\,A\,X = A \Rightarrow X & \\
[\![\mathsf{fn.tm}]\!] = \langle\text{canonical}\rangle & [\![\mathsf{case}]\!]\,u\,f\,g = \begin{cases} f(x) & \text{if } u = \mathsf{inl}\,x \\ g(x) & \text{if } u = \mathsf{inr}\,x \end{cases}
\end{array}$$

Note that the coproduct $A + B$ above is computed in the category of predomains[4] and need not be preserved by the embedding into $\boldsymbol{\mathcal{S}}_{\mathsf{C}}$. We next add the security levels and the sealing modality, interpreted as the pushout of predomains $\langle l\rangle \bullet A$, again computed in the category of predomains. We define the unsealing operator for $B : [\![\mathsf{tp}^{+}_{\bullet\langle l\rangle}]\!]$ using the universal property of the pushout.

$$\begin{array}{ll}
[\![\langle l\rangle]\!] = \langle l\rangle = \boldsymbol{p}_{\mathsf{C}}^{*}\mathsf{y}_{\mathcal{P}}l & [\![\mathsf{unseal}_l]\!]\,u\,f = \\
[\![\mathsf{T}_l]\!]\,A = \langle l\rangle \bullet A & \quad \begin{cases} f x & \text{if } u = \eta_{\bullet\langle l\rangle}\,x \\ \bigstar & \text{if } u = \bigstar \end{cases} \\
[\![\mathsf{seal}_l]\!]\,a = \eta_{\bullet\langle l\rangle}\,a &
\end{array}$$

▶ **Observation 19.** *Morphisms $\langle l\rangle \bullet A \longrightarrow B$ are in bijective correspondence with morphisms $A \longrightarrow B$ that restricts to a* weakly constant *function under $\langle l\rangle$.*

We may now interpret the termination declassification operation. Fixing a sealed type $A : [\![\mathsf{tp}^{+}_{\bullet\langle l\rangle}]\!]$, we must define the dotted lift below using the universal property of the pushout and the action of the stabilizer of $A$ on $\mathsf{L}A$, noting that $\langle l\rangle \in \mathsf{Stab}_{\Sigma^\vee}\,A$ by assumption:

$$\begin{array}{c}
A \xrightarrow{\ \eta_A\ } \mathsf{L}A \\
\eta_{\bullet\langle l\rangle} \circ \eta_A \downarrow \quad \nearrow_{[\![\mathsf{tdcl}_{\langle l\rangle}]\!]} \\
\langle l\rangle \bullet \mathsf{L}A
\end{array}
\qquad
\begin{array}{l}
[\![\mathsf{tdcl}_{\langle l\rangle}]\!]\,u = \\
\quad \begin{cases} \langle l\rangle \parallel_{\mathsf{L}A}\,x & \text{if } u = \eta_{\bullet\langle l\rangle}\,x \\ \langle l\rangle \parallel_{\mathsf{L}A}\,\bot & \text{if } u = \bigstar \end{cases}
\end{array}$$

To see that the above is well-defined, we observe that under $\langle l\rangle$ both branches return the (unique) computation whose termination support is $\langle l\rangle$. With this definition, the required computation rule holds by virtue of Lemma 18.

## 5.5 Noninterference in the denotational semantics

▶ **Definition 20.** *A function $u : A \longrightarrow B$ is called **weakly constant** [26] if for all $x, y : A$ we have $u\,x = u\,y$. A partial function $u : A \longrightarrow \mathsf{L}B$ is called **partially constant** if for all $x, y : A$ such that $u\,x{\downarrow} \wedge u\,y{\downarrow}$, we have $u\,x = u\,y$.*

[4] Any reflective subcategory of a cocomplete category is cocomplete: first compute the colimit in the outer category, and then apply the reflection.

For the following, let $l \in \mathcal{P}$ be a security level.

▶ **Lemma 21.** *Let $A$ be a $\langle l \rangle$-sealed predomain and let $B$ be a $\langle l \rangle$-transparent predomain; then (1) any function $A \longrightarrow B$ is weakly constant, and (2) any partial function $A \longrightarrow \mathsf{L}B$ is partially constant.*

The following lemma follows from Axiom SDT-5.

▶ **Lemma 22.** *The predomain $[\![\mathsf{bool}]\!]$ is $\langle l \rangle$-transparent.*

In order for Lemma 21 to have any import as far as the equational theory is concerned, we must establish computational adequacy. This is the topic of Section 6.

# 6 Adequacy of the denotational semantics

We must argue that the denotational semantics agrees with the theory as far as convergence and return values is concerned. We do so using a Plotkin-style logical relations argument, phrased in the language of Synthetic Tait Computability [48, 50, 49].

## 6.1 Synthetic Tait computability of formal approximation

In this section we will work abstractly with a Grothendieck topos $\mathbf{G}$ satisfying several axioms that will make it support a Kripke logical relation for adequacy.

▶ **Notation 23.** For each universe $\mathbf{U} \in \mathcal{S}_{\mathbf{G}}$ there is a type $\mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{U}}$ of internal $\mathfrak{T}$-algebras whose type components are valued in $\mathbf{U}$. $\mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{U}}$ is a dependent record containing a field for every constant in the signature by which we generated $\mathfrak{T}$. Assuming enough universes, functors $\mathfrak{T} \longrightarrow \mathcal{S}_{\mathbf{G}}/E$ correspond up to isomorphism to morphisms $E \longrightarrow \mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{U}}$. This is the relationship between the internal language and the *functorial semantics* à la Lawvere [27].

▶ **Axiom STC-1.** *There are two disjoint propositions $\mathrm{T}, \mathrm{C} \in \mathcal{O}_{\mathbf{G}}$ such that $\mathrm{T} \wedge \mathrm{C} = \bot$. We will refer to these as the **syntactic** and **computational phases** respectively. We will write $\mathrm{B} = \mathrm{T} \vee \mathrm{C}$ for the disjoint union of the two phases.*

▶ **Axiom STC-2.** *Within the syntactic phase, there exists a $\mathfrak{T}$-algebra $\mathcal{A}^{\mathrm{T}} : \mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{U}_{/\mathrm{T}}}$ such that the corresponding functor $\mathfrak{T} \longrightarrow \mathcal{S}_{\mathbf{G}}/\mathrm{T}$ is fully faithful.*

▶ **Axiom STC-3.** *Within the computational phase, the axioms of $\mathcal{P}$-indexed synthetic domain theory (Axioms SDT-1–SDT-5) are satisfied.*

As a consequence of Axiom STC-3, we have a *computational* $\mathfrak{T}$-algebra $\mathcal{A}^{\mathrm{C}} : \mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{U}_{/\mathrm{C}}}$ given by the constructions of Section 5.4. Gluing together the two models $\mathcal{A}^{\mathrm{T}}, \mathcal{A}^{\mathrm{C}}$ we see that $\mathbf{G}_{/\mathrm{B}}$ supports a model $\mathcal{A}^{\mathrm{B}} = [\mathrm{T} \hookrightarrow \mathcal{A}^{\mathrm{T}}, \mathrm{C} \hookrightarrow \mathcal{A}^{\mathrm{C}}]$ of $\mathfrak{T}$. The final Axiom STC-4 above is needed in the approximation structure of $\mathsf{tdcl}_{\langle l \rangle}$.

▶ **Axiom STC-4.** *For each $l \in \mathcal{P}$ we have $\mathcal{A}^{\mathrm{C}}.\langle l \rangle \leq \mathrm{B} \bullet \mathcal{A}^{\mathrm{T}}.\langle l \rangle$.*

▶ **Theorem 24.** *There exists a topos $\mathbf{G}$ satisfying Axioms STC-1–STC-4 containing open subtopoi $\mathbf{G}_{/\mathrm{T}} = \widehat{\mathfrak{T}}$ and $\mathbf{G}_{/\mathrm{C}} = \mathbf{C}$ such that the complementary closed subtopos is $\mathbf{G}_{\bullet\mathrm{B}} = \mathbf{P}$.*

**Proof.** We may construct a topos using a variant of the Artin gluing construction of Sterling and Harper [50], which we detail in our extended version. ◀

By Axioms STC-1 and STC-2, any such topos **G** supports a model of the ***synthetic Tait computability*** of Sterling and Harper [50, 48]. In the internal language of $\mathcal{S}_{\mathsf{G}}$, the phase B induces a pair of complementary transparency/open and sealing/closed modalities that can be used to synthetically construct formal approximation relations in the sense of Plotkin between computational objects and syntactical objects. Viewing an object $E \in \mathcal{S}_{\mathsf{G}}$ as a family $x : \textsc{c} \Rightarrow E, x' : \textsc{t} \Rightarrow E \vdash \{E \mid \textsc{c} \hookrightarrow x, \textsc{t} \hookrightarrow x'\}$ of B-sealed types over the B-transparent type $(\textsc{b} \Rightarrow E) \cong ((\textsc{c} \Rightarrow E) \times (\textsc{t} \Rightarrow E))$, we may think of $E$ as a *proof-relevant* formal approximation relation between its computational and syntactic parts, which we might term a "formal approximation structure".

▶ **Notation 25** (Extension types). We recall ***extension types*** from Riehl and Shulman [40]. Given a proposition $\phi : \Omega$ and a partial element $e : \phi \Rightarrow E$, we will write $\{E \mid \phi \hookrightarrow e\}$ for the collection of elements of $E$ that restrict to $e$ under $\phi$, *i.e.* the subobject $\{x : E \mid \phi \Rightarrow (x = e)\} \rightarrowtail E$. Note that $\{E \mid \phi \hookrightarrow e\}$ is always $\phi$-sealed, since it becomes the singleton type $\{e\}$ under $\phi$.

Each universe $\mathbf{U}$ of $\mathcal{S}_{\mathsf{G}}$ satisfies a remarkable *strictification* property with respect to any proposition $\phi : \Omega$ that allows one to construct codes for dependent sums of families of $\phi$-sealed types over a $\phi$-transparent type in such a way that they restrict *exactly* to the $\phi$-transparent part under $\phi$. This refinement of dependent sums is called a ***strict glue type***:[5]

STRICT GLUE TYPES

$$\frac{A : \phi \Rightarrow \mathbf{U} \qquad B : ((z : \phi) \Rightarrow A\, z) \to \mathbf{U} \qquad \forall x.\ \mathsf{isSealed}_{\phi}\,(B, x)}{\begin{gathered}(x : A) \ltimes_{\phi} B\, x : \{\mathbf{U} \mid z : \phi \hookrightarrow A\, z\} \\ \mathsf{glue}_{\phi} : \{((x : (z : \phi) \Rightarrow A\, z) \times B\, x) \cong (x : A) \ltimes_{\phi} B\, x \mid \phi \hookrightarrow \pi_1\}\end{gathered}}$$

▶ **Notation 26** (Strict glue types). We impose two notations assuming $A, B$ as above. Given $a : (z : \phi) \Rightarrow A\, z$ and $b : B\, a$, we write $\mathsf{glue}\,[b \mid \phi \hookrightarrow a]$ for $\mathsf{glue}_{\phi}(a, b)$. Given $g : (x : A) \ltimes_{\phi} B\, x$, we write $\mathsf{unglue}_{\phi} g : B\, g$ for the element $\pi_2\left(\mathsf{glue}_{\phi}^{-1}\, g\right)$.

▶ **Notation 27.** Let $E$ be a type in $\mathcal{S}_{\mathsf{G}}$ and fix elements $e : \textsc{c} \Rightarrow E$ and $e' : \textsc{t} \Rightarrow E$ of the computational and syntactical parts of $E$ respectively; we will write $e \lhd_E e'$, pronounced "$e$ formally approximates $e'$", for the extension type $\{E \mid \textsc{c} \hookrightarrow e, \textsc{t} \hookrightarrow e'\}$.

This is the connection between synthetic Tait computability and analytic logical relations; the open parts of an object correspond to the *subjects* of a logical relation and the closed parts of an object correspond to the evidence of that relation.

▶ **Definition 28** (Formal approximation relations). *A type $E$ is called a* ***formal approximation relation*** *when for any* B*-point $e : \textsc{b} \Rightarrow E$, the extension type $\{E \mid \textsc{b} \hookrightarrow e\}$ is a proposition,* i.e. *any two elements of $e \lhd_E e$ are equal.*

We will write $\mathsf{Rel}_{\mathbf{U}} \subseteq \mathbf{U}$ for the subuniverse of formal approximation relations.

▶ **Definition 29** (Admissible formal approximation relations). *Let $E$ be a formal approximation relation such that $\textsc{c} \Rightarrow E$ is a predomain equipped with an $\mathsf{L}$-algebra structure. We say that $E$ is* ***admissible*** *at $x : \textsc{t} \Rightarrow E$ when the subobject $\{E \mid \textsc{t} \hookrightarrow x\} \subseteq \textsc{c} \Rightarrow E$ is admissible in the sense of synthetic domain theory,* i.e. *contains $\perp$ and is closed under formal suprema of formal $\boldsymbol{\omega}$-chains. We say that $E$ is admissible when it is admissible at every such $x$.*

[5] In presheaves, the universes of Hofmann and Streicher [21, 51] satisfy this property directly; for sheaves, there is an alternative transfinite construction of universes enjoying this property [19]. Our presentation in terms of transparency and sealing is an equivalent reformulation of the strictness property identified by several authors in the context of the semantics of homotopy type theory [24, 52, 46, 7, 32, 5, 47, 4].

▶ **Lemma 30** (Scott induction). *Let $X$ be a formal approximation relation such that $\textsc{c} \Rightarrow X$ is a domain. Let $f : X \to X$ be an endofunction on $X$ and let $x : \textsc{t} \Rightarrow X$ be a syntactical fixed point of $f$ in the sense that $\textsc{t} \Rightarrow (x = f\,x)$; if $X$ is admissible at $x$, then we have $\mathsf{fix}\,f \lhd_X x$.*

Our goal can be rephrased now in the internal language; choosing a universe $\mathbf{V} \supset \mathbf{U}$, we wish to define a suitable $\mathbf{V}$-valued algebra $\mathcal{A} \in \mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{V}}$ that restricts under $\textsc{b}$ to $\mathcal{A}^{\textsc{b}}$, *i.e.* an element $\mathcal{A} \in \{\mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{V}} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}\}$. This can be done quite elegantly in the internal language of $\boldsymbol{\mathcal{S}}_{\mathbf{G}}$, *i.e.* the *synthetic Tait computability of formal approximation structures.* The high-level structure of our model construction is summarized as follows:

> We interpret value types as ***formal approximation structures*** over a syntactic value type and a predomain; we interpret computation types as ***admissible formal approximation relations*** between a syntactic computation type and a domain.

To make this precise, we will define $\mathcal{A}.\mathsf{tp}^{+} \in \{\mathbf{V} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{tp}^{+}\}$ as the collection of types that restrict to an element of $\mathcal{A}^{\textsc{t}}.\mathsf{tp}^{+}$ in the syntactic phase and to an element of $\mathcal{A}^{\textsc{c}}.\mathsf{tp}^{+} = \mathbf{Predom}$ in the computational phase. This is achieved using strict gluing:

$$\mathcal{A}.\mathsf{tp}^{+} = \left(A : \mathcal{A}^{\textsc{b}}.\mathsf{tp}^{+}\right) \ltimes_{\textsc{b}} \{\mathbf{U} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{tm}\,A\} \qquad \mathcal{A}.\mathsf{tm} = \mathsf{unglue}_{\textsc{b}}$$

The above is well-defined because $\mathcal{A}^{\textsc{b}}.\mathsf{tp}^{+}$ is $\textsc{b}$-transparent and $\{\mathbf{U} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{tm}\,A\}$ is $\textsc{b}$-sealed. We also have $\textsc{t} \Rightarrow \mathcal{A}.\mathsf{tp}^{+} = \mathcal{A}^{\textsc{t}}.\mathsf{tp}^{+}$ and $\textsc{c} \Rightarrow \mathcal{A}.\mathsf{tp}^{+} = \mathbf{Predom}$. Next we define the formal approximation structure of computation types:

$$\mathcal{A}.\mathsf{tp}^{\ominus} = \left(X : \mathcal{A}^{\textsc{b}}.\mathsf{tp}^{\ominus}\right) \ltimes_{\textsc{b}} \{X' : \{\mathsf{Rel}_{\mathbf{U}} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{tm}\left(\mathcal{A}^{\textsc{b}}.\mathsf{U}\,X\right)\} \mid X' \text{ is admissible}\}$$

To see that the above is well-defined, we must check that the family component of the gluing is pointwise $\textsc{b}$-sealed, which follows because the property of being admissible is $\textsc{b}$-sealed. To see that this is the case, we observe that it is obviously $\textsc{t}$-sealed and also (less obviously) $\textsc{c}$-sealed: under $\textsc{c}$, $X'$ restricts to the "total" predicate on $X$ which is always admissible. To define the thunking connective, we simply forget that a given admissible approximation relation was admissible: $\mathcal{A}.\mathsf{U}\,X = \mathsf{glue}\left[\mathsf{unglue}_{\textsc{b}}X \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{U}\,X\right]$. To interpret free computation types, we proceed in two steps; first we define the formal approximation relation as an element of $\mathsf{Rel}_{\mathbf{U}}$ and then we glue it onto syntax and semantics.

$$[\mathsf{F}]\,A = \left(u : \mathcal{A}^{\textsc{b}}.(\mathsf{UF})\,A\right) \ltimes_{\textsc{b}} (\textsc{c} \Rightarrow u{\downarrow}) \Rightarrow \textsc{b} \bullet \exists a : A.\textsc{b} \Rightarrow u = \mathcal{A}^{\textsc{b}}.\mathsf{ret}\,a$$
$$\mathcal{A}.\mathsf{F}\,A = \mathsf{glue}\left[[\mathsf{F}]\,A \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{F}\right]$$

In simpler language, we have $u \lhd_{[\mathsf{F}]\,A} v$ if and only if $v$ terminates syntactically whenever $u$ terminates such that the value of $u$ formally approximates the value of $v$. This is the standard clause for lifting in an adequacy proof, phrased in synthetic Tait computability. ; the use of the sealing modality is an artifact of synthetic Tait computability, ensuring that the relation is pointwise $\textsc{b}$-sealed. The $\mathsf{ret}, \mathsf{bind}$ operations are easily shown to preserve the formal approximation relations. The construction of formal approximation structures for product and function spaces is likewise trivial. Using Scott induction (Lemma 30) we can show that fixed points also lie in the formal approximation relations; we elide the details. Next we deal with the information flow constructs, starting by interpreting each security policy $\mathcal{A}.\langle l\rangle$ as $\mathcal{A}^{\textsc{b}}.\langle l\rangle$. The sealing modality is interpreted below:

$$[\mathsf{T}_l]\,A = \left(u : \mathcal{A}^{\textsc{b}}.\mathsf{T}_l A\right) \ltimes_{\textsc{b}} \textsc{b} \bullet \mathcal{A}.\langle l\rangle \bullet \{a : A \mid \textsc{b} \Rightarrow u = \mathcal{A}^{\textsc{b}}.\mathsf{seal}_l a\}$$
$$\mathcal{A}.\mathsf{T}_l A = \mathsf{glue}\left[[\mathsf{T}_l]\,A \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}.\mathsf{T}_l\right]$$

▶ **Theorem 31** (Fundamental theorem of logical relations). *The preceding constructions arrange into an algebra $\mathcal{A} \in \{\mathfrak{T}\text{-}\mathbf{Alg}_{\mathbf{V}} \mid \textsc{b} \hookrightarrow \mathcal{A}^{\textsc{b}}\}$.*

## 6.2 Adequacy and syntactic noninterference results

The following definitions and results in this section are global rather than internal. We may immediately read off from the logical relation of Section 6.1 a few important properties relating value terms and their denotations. The results of this section depend heavily on the assumption that the functor $\mathfrak{T} \hookrightarrow \boldsymbol{\mathcal{S}}_{\mathsf{G}}/\mathrm{T}$ is fully faithful (Axiom STC-2).

▶ **Theorem 32** (Value adequacy). *For any closed values $u, v : \mathbf{1}_{\mathfrak{T}} \longrightarrow \mathsf{bool}$, we have $[\![u]\!] = [\![v]\!]$ if and only if $u \equiv_{\mathsf{bool}} v$; moreover we have either $u \equiv_{\mathsf{bool}} \mathsf{tt}$ or $u \equiv_{\mathsf{bool}} \mathsf{ff}$.*

Let $u : \mathbf{1}_{\mathfrak{T}} \longrightarrow \mathsf{UF}A$ be a closed computation.

▶ **Definition 33** (Convergence and divergence). *We say that $u$ **converges** when there exists $a : \mathbf{1}_{\mathfrak{T}} \longrightarrow A$ such that $u = \mathsf{ret}\, a$. Conversely, we say that $u$ **diverges** when there does not exist such an $a$. We will write $u\Downarrow$ to mean that $u$ converges, and $u\Uparrow$ to mean that $u$ diverges.*

▶ **Theorem 34** (Computational adequacy). *The computation $u$ converges iff $[\![u]\!]\downarrow = \top$.*

▶ **Theorem 35** (Termination-insensitive noninterference). *Let $A$ be a syntactic type such that $\mathsf{sealed}_{\langle l \rangle}\, A$ holds; fix a term $c : A \longrightarrow \mathsf{UF}\,\mathsf{bool}$. Then for all $x, y : \mathbf{1}_{\mathfrak{T}} \longrightarrow A$ such that $c\,x\Downarrow$ and $c\,y\Downarrow$, we have $c\,x \equiv_{\mathsf{UF}\,\mathsf{bool}} c\,y$.*

We give an example of a program whose termination behavior hinges on a classified bit to demonstrate that our noninterference result is non-trivial.

▶ **Example 36.** There exists an $\langle l \rangle$-sealed type $A$ and a term $c : A \longrightarrow \mathsf{UF}\,\mathsf{unit}$ such that for some $x, y : \mathbf{1}_{\mathfrak{T}} \longrightarrow A$ we have $c\,x\Downarrow$ and yet $c\,y\Uparrow$.

**Proof.** Choose $A :\equiv \mathsf{T}_l\,\mathsf{bool}$ and consider the following terms:

$$\top :\equiv \mathsf{ret}\,() \quad \bot :\equiv \mathsf{fix}\,(\lambda z.z) \quad x :\equiv \mathsf{seal}_l\,\mathsf{tt} \quad y :\equiv \mathsf{seal}_l\,\mathsf{ff}$$
$$c :\equiv \lambda u.\,\mathsf{tdcl}_{\langle l \rangle}\,(\mathsf{unseal}_l\,u\,(\lambda b.\,\mathsf{seal}_l\,(\mathsf{if}\,b\,\top\,\bot)))$$

We then have $c\,x \equiv_{\mathsf{UF}\,\mathsf{unit}} \top$ and therefore $c\,x\Downarrow$. On the other hand, we have $c\,y \equiv_{\mathsf{UF}\,\mathsf{unit}} \mathsf{tdcl}_{\langle l \rangle}\,(\mathsf{seal}_l\,\bot)$; executing the denotational semantics, we have $[\![c\,y]\!]\downarrow = \langle l \rangle$. From the full and faithfulness assumption of Axiom SDT-4, we know that $\langle l \rangle$ is not globally equal to $\top$; hence we conclude from Theorem 34 that $c\,y\Uparrow$. ◀

## References


1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, San Antonio, Texas, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292555.
2. Mathieu Anel and André Joyal. Topo-logie. In Mathieu Anel and Gabriel Catren, editors, *New Spaces in Mathematics: Formal and Conceptual Reflections*, volume 1, chapter 4, pages 155–257. Cambridge University Press, 2021. doi:10.1017/9781108854429.007.
3. Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. *Théorie des topos et cohomologie étale des schémas*, volume 269, 270, 305 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1972. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat.
4. Steve Awodey. A Quillen model structure on the category of cartesian cubical sets. Unpublished notes, 2021. URL: https://github.com/awodey/math/blob/e8c715cc5cb6a966e736656bbe54d0483f9650fc/QMS/qms.pdf.

5 Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2016.23.

6 William J. Bowman and Amal Ahmed. Noninterference for free. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 101–113. Association for Computing Machinery, 2015. doi:10.1145/2784731.2784733.

7 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. *IfCoLog Journal of Logics and their Applications*, 4(10):3127–3169, November 2017. arXiv:1611.02108.

8 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, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2021.28.

9 M. Fiore, G. Plotkin, and J. Power. Complete cuboidal sets in axiomatic domain theory. In *Logic in Computer Science, Symposium on*, page 268, Los Alamitos, CA, USA, July 1997. IEEE Computer Society. doi:10.1109/LICS.1997.614954.

10 Marcelo Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, November 1994. URL: https://era.ed.ac.uk/handle/1842/406.

11 Marcelo P. Fiore. An enrichment theorem for an axiomatisation of categories of domains and continuous functions. *Mathematical Structures in Computer Science*, 7(5):591–618, October 1997. doi:10.1017/S0960129597002429.

12 Marcelo P. Fiore and Gordon D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic, 10th International Workshop, CSL '96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*, volume 1258 of *Lecture Notes in Computer Science*, pages 129–149. Springer, 1996. doi:10.1007/3-540-63172-0_36.

13 Marcelo P. Fiore and Giuseppe Rosolini. The category of cpos from a synthetic viewpoint. In Stephen D. Brookes and Michael W. Mislove, editors, *Thirteenth Annual Conference on Mathematical Foundations of Progamming Semantics, MFPS 1997, Carnegie Mellon University, Pittsburgh, PA, USA, March 23-26, 1997*, volume 6 of *Electronic Notes in Theoretical Computer Science*, pages 133–150. Elsevier, 1997. doi:10.1016/S1571-0661(05)80165-3.

14 Marcelo P. Fiore and Giuseppe Rosolini. Two models of synthetic domain theory. *Journal of Pure and Applied Algebra*, 116(1):151–162, 1997. doi:10.1016/S0022-4049(96)00164-8.

15 Marcelo P. Fiore and Giuseppe Rosolini. Domains in H. *Theoretical Computer Science*, 264(2):171–193, August 2001. doi:10.1016/S0304-3975(00)00221-8.

16 Peter Freyd. On proving that **1** is an indecomposable projective in various free categories. Unpublished manuscript, 1978.

17 Daniel Gratzer. Normalization for multimodal type theory. To appear, *Symposium on Logic in Computer Science Logic (LICS) '22*, 2021. arXiv:2106.01414.

18 Daniel Gratzer and Lars Birkedal. A stratified approach to Löb induction. To appear, *International Conference on Formal Structures for Computation and Deduction (FSCD) '22*, 2022. URL: https://jozefg.github.io/papers/a-stratified-approach-to-lob-induction.pdf.

19 Daniel Gratzer, Michael Shulman, and Jonathan Sterling. Strict universes for Grothendieck topoi. Unpublished manuscript, February 2022. doi:10.48550/arXiv.2202.12012.

20 Daniel Gratzer and Jonathan Sterling. Syntactic categories for dependent type theory: sketching and adequacy. Unpublished manuscript, 2020. arXiv:2012.10783.

**21** Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished note, 1997. URL: `https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf`.

**22** J. M. E. Hyland. First steps in synthetic domain theory. In Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini, editors, *Category Theory*, pages 131–156, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

**23** Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2*. Number 43 in Oxford Logical Guides. Oxford Science Publications, 2002.

**24** Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23:2071–2126, March 2021. `doi:10.4171/JEMS/1050`.

**25** G. A. Kavvos. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019. `doi:10.1145/3290333`.

**26** Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory. *Logical Methods in Computer Science*, 13(1):1–36, March 2017. `doi:10.23638/LMCS-13(1:15)2017`.

**27** F. William Lawvere. Functorial Semantics of Algebraic Theories. *Reprints in Theory and Applications of Categories*, 4:1–121, 2004. URL: `http://tac.mta.ca/tac/reprints/articles/5/tr5.pdf`.

**28** F. William Lawvere. Axiomatic cohesion. *Theory and Applications of Categories*, 19:41–49, June 2007. URL: `http://www.tac.mta.ca/tac/volumes/19/3/19-03.pdf`.

**29** Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

**30** Cristina Matache, Sean Moss, and Sam Staton. Recursion and Sequentiality in Categories of Sheaves. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.FSCD.2021.25`.

**31** Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022. `doi:10.1145/3498670`.

**32** Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2016.24`.

**33** Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, December 1976. `doi:10.1007/BF00268134`.

**34** Wesley Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, University of Edinburgh, July 1991.

**35** Bernhard Reus. *Program Verification in Synthetic Domain Theory*. PhD thesis, Ludwig-Maximilians-Universität München, München, November 1995.

**36** Bernhard Reus. Synthetic domain theory in type theory: Another logic of computable functions. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 363–380, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. `doi:10.1007/BFb0105416`.

**37** Bernhard Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23(3):411–444, 1999. `doi:10.1023/A:1006258506401`.

**38** Bernhard Reus and Thomas Streicher. Naïve synthetic domain theory — a logical approach. Unpublished manuscript, September 1993.

**39** Bernhard Reus and Thomas Streicher. General synthetic domain theory — a logical approach. *Mathematical Structures in Computer Science*, 9(2):177–223, 1999. `doi:10.1017/S096012959900273X`.

40 Emily Riehl and Michael Shulman. A type theory for synthetic $\infty$-categories. *Higher Structures*, 1:147–224, 2017. arXiv:1705.07442.

41 Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *Logical Methods in Computer Science*, Volume 16, Issue 1, January 2020. doi:10.23638/LMCS-16(1:2)2020.

42 Guiseppe Rosolini. *Continuity and effectiveness in topoi*. PhD thesis, University of Oxford, 1986.

43 Patrick Schultz and David I. Spivak. *Temporal Type Theory*, volume 29 of *Progress in Computer Science and Applied Logic*. Birkhäuser Basel, 2019. doi:10.1007/978-3-030-00704-1.

44 Michael Shulman. Localization as an inductive definition, December 2011. URL: https://homotopytypetheory.org/2011/12/06/inductive-localization/.

45 Michael Shulman. Reflective subfibrations, factorization systems, and stable units, December 2011. URL: https://golem.ph.utexas.edu/category/2011/12/reflective_subfibrations_facto.html.

46 Michael Shulman. The univalence axiom for elegant reedy presheaves. *Homology, Homotopy and Applications*, 17:81–106, 2015. doi:10.4310/HHA.2015.v17.n2.a6.

47 Michael Shulman. All $(\infty, 1)$-toposes have strict univalent universes. Unpublished manuscript, April 2019. arXiv:1904.07004.

48 Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2021. CMU technical report CMU-CS-21-142. doi:10.5281/zenodo.5709838.

49 Jonathan Sterling and Carlo Angiuli. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15, Los Alamitos, CA, USA, July 2021. IEEE Computer Society. doi:10.1109/LICS52264.2021.9470719.

50 Jonathan Sterling and Robert Harper. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM*, 68(6), October 2021. doi:10.1145/3474834.

51 Thomas Streicher. Universes in toposes. In Laura Crosilla and Peter Schuster, editors, *From Sets and Types to Topology and Analysis: Towards practical foundations for constructive mathematics*, volume 48 of *Oxford Logical Guides*, pages 78–90. Oxford University Press, Oxford, 2005. doi:10.1093/acprof:oso/9780198566519.001.0001.

52 Thomas Streicher. A model of type theory in simplicial sets: A brief introduction to voevodsky's homotopy type theory. *Journal of Applied Logic*, 12(1):45–49, 2014. doi:10.1016/j.jal.2013.04.001.

53 Paul Taylor. The fixed point property in synthetic domain theory. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 152–160, 1991. doi:10.1109/LICS.1991.151640.

54 Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 115–125, Snow Bird, UT, USA, 2004. Association for Computing Machinery. doi:10.1145/1016850.1016868.

55 Steven Vickers. Locales and toposes as spaces. In Marco Aiello, Ian Pratt-Hartmann, and Johan Van Benthem, editors, *Handbook of Spatial Logics*, pages 429–496. Springer Netherlands, Dordrecht, 2007. doi:10.1007/978-1-4020-5587-4_8.

# Combined Hierarchical Matching: the Regular Case

**Serdar Erbatur**
University of Texas at Dallas, TX, USA

**Andrew M. Marshall**
University of Mary Washington, Fredericksburg, VA, USA

**Christophe Ringeissen**
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

## Abstract

Matching algorithms are often central sub-routines in many areas of automated reasoning. They are used in areas such as functional programming, rule-based programming, automated theorem proving, and the symbolic analysis of security protocols. Matching is related to unification but provides a somewhat simplified problem. Thus, in some cases, we can obtain a matching algorithm even if the unification problem is undecidable. In this paper we consider a hierarchical approach to constructing matching algorithms. The hierarchical method has been successful for developing unification algorithms for theories defined over a constructor sub-theory. We show how the approach can be extended to matching problems which allows for the development, in a modular way, of hierarchical matching algorithms. Here we focus on regular theories, where both sides of each equational axiom have the same set of variables. We show that the combination of two hierarchical matching algorithms leads to a hierarchical matching algorithm for the union of regular theories sharing only a common constructor sub-theory.

**2012 ACM Subject Classification** Theory of computation → Equational logic and rewriting; Theory of computation → Automated reasoning

**Keywords and phrases** Matching, combination problem, equational theories

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.6

**Acknowledgements** We would like to thank the reviewers for their comments that were very helpful to improve the readability of the paper.

## 1 Introduction

Matching procedures play a central role in automated reasoning and in various declarative programming paradigms such as functional programming or (constraint) logic programming. For example, in rule-based programming [9, 11], matching is needed to apply a rule and thus to perform computations. In automated theorem proving [1, 7], matching is useful to simplify existing facts via contraction inferences. For the verification of security protocols, dedicated provers [8, 20, 25] handle protocols specified in a symbolic way. In these reasoning tools, the capabilities of an intruder are modeled using equational theories, and the reasoning is supported by decision procedures and solvers modulo equational theories, including matching and unification. An equational matching problem is an equational unification problem with free constants where each equation has a ground side. This particular form of equational unification with free constants remains undecidable in general. However, the successful application of equational rewriting in rule-based programming languages [9, 11, 26] has demonstrated the usefulness of developing matching algorithms for particular equational theories such as Associativity ($A$), Commutativity ($C$) or Associativity-Commutativity ($AC$). In many practical applications, the underlying equational theory is defined as a union of theories, like a union of $AC$-symbols. In that case, it is quite natural to solve the matching

© Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 6; pp. 6:1–6:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

problem for the union of theories in a modular way by combining the matching algorithms available for the component theories of the union. For unification and matching, there are terminating and complete combination procedures for the union of signature-disjoint theories [34, 3]. These combination procedures can be extended to some non-disjoint unions of theories sharing only constructor symbols, but it is quite difficult to find particular cases where these procedures terminate [12], although several terminating cases have been identified [30, 5, 13, 18, 19]. Here, matching being a restricted form of unification can be helpful since matching can be considered as a simpler problem. For example, $A$-matching is finitary, that is, the set of solutions of an $A$-matching problem is finite, whereas $A$-unification is infinitary. Thus for matching, we may be able to show termination even if we cannot in unification.

In this paper we consider the matching problem in theories $F \cup E$ where $E$ is a constructor sub-theory for $F \cup E$, $F$ being called an $E$-constructed theory. We show how to apply the relatively recently developed hierarchical combination approach [14, 13, 18, 19] to build $F \cup E$-matching algorithms. We focus on regular theories for $F \cup E$, where both sides of each equational axiom have the same set of variables. This is a natural assumption since any matching problem has only ground solutions in regular theories. We adopt a new modular definition of $E$-constructed theory [18] and consider the class of regular theories $F \cup E$ where $F$ is $E$-constructed. This class is closed by any union of theories $F_1 \cup E$ and $F_2 \cup E$ sharing only symbols in $E$. We show that combining hierarchical matching algorithms known for $F_1 \cup E$ and $F_2 \cup E$ leads to a hierarchical matching algorithm for the union of $F_1 \cup E$ and $F_2 \cup E$. In our hierarchical matching approach, we consider a new type of layer-reduced term mappings that can be constructed in a modular way to reduce the theory layers of any ground term occurring in a matching problem. In addition, we also show how the hierarchical approach can be used for solving the $F \cup E$-equality of terms in layer-reduced form, required by a hierarchical $F \cup E$-matching algorithm. The presented hierarchical approach applies to the important case $R \cup E$ where $(R, E)$ is any $E$-convergent term rewrite system (TRS for short) where all the symbols in $E$ are constructors, called $E$-constructed TRS. It applies also to theories $F \cup E$ where $F$ is $E$-constructed and $F \cup E$ is a finite syntactic theory [28, 23]. In that case the underlying hierarchical algorithm can be simply expressed using some additional mutation rules generalizing the very classical decomposition rule used in syntactic unification. A form of syntacticness can also be applied to $E$-constructed TRSs which are *innermost-resolvent*, exemplified by distributive theories and exponentiation theories.

**Motivating Example from Security Protocols.** Modular exponentiation is a common operation found in many theories modeling security protocols [24]. For example, exponentiation with a multiplication operator can be modeled with the following axioms $\{e(e(x,y),z) = e(x, y * z),\ e(x * y, z) = e(x,z) * e(y,z)\}$ and the $AC$ theory for $*$. Obtaining unification algorithms for this exponentiation theory (and related theories) has proven difficult. In fact, it is undecidable in the case where $*$ is $AC$ [27]. Because of this difficulty, the theory is often changed to include a new operator $\circledast$, and a modification of the first axiom to $e(e(x,y),z) = e(x, y \circledast z)$. Thus, creating two multiplication operators rather than one. Even in this case, obtaining a unification algorithm is not always possible with several undecidability results having been shown depending on the properties of $*$ and $\circledast$ [22]. However, by using the modular combination result developed in this paper, we can obtain a hierarchical matching algorithm for the exponentiation theories and more. The modular aspect to the combination algorithm is also attractive since we can reuse a matching algorithm for the base theory, $AC$ in this example.

**Outline.** After this introduction and the next section on preliminaries, the paper is organized as follows. Sections 3 and 4 present the different classes of theories $F \cup E$ considered in the paper, and some modularity results we can obtain for the problems of $F \cup E$-equality and of $F \cup E$-matching. The class of $E$-constructed theories is introduced in Section 3, while Section 4 focuses on $E$-constructed theories admitting mutation-based matching algorithms. In Section 5, we present our notion of hierarchical $F \cup E$-matching algorithm. Our results on combining hierarchical $F \cup E$-matching algorithms are shown in Section 6. In addition, Section 7 shows that our methodology can be applied to get hierarchical decision procedures for the $F \cup E$-equality. Related work and concluding remarks are discussed in Section 8. Appendix A includes omitted proofs.

## 2 Preliminaries

We use the standard notation of equational unification [4] and term rewriting systems [2]. Given a first-order signature $\Sigma$ and a (countable) set of variables $V$, the set of $\Sigma$-terms over variables $V$ is denoted by $T(\Sigma, V)$. Given a (countable) set of constants $C$ disjoint from $V$ and $\Sigma$, the set of $\Sigma$-terms over $V \cup C$ is denoted in the same way by $T(\Sigma, V \cup C)$. In the following, a $\Sigma$-term is assumed to be a term in $T(\Sigma, V \cup C)$. The set of variables (resp., constants) from $V$ (resp., $C$) occurring in a term $t \in T(\Sigma, V \cup C)$ is denoted by $Var(t)$ (resp., $Cst(t)$). A term $t$ is *ground* if $Var(t) = \emptyset$. A $\Sigma \cup C$-rooted term is a term whose root symbol is in $\Sigma \cup C$. For any position $p$ in a term $t$ (including the root position $\epsilon$), $t(p)$ is the symbol at position $p$, $t|_p$ is the subterm of $t$ at position $p$, and $t[u]_p$ is the term $t$ in which $t|_p$ is replaced by $u$. A substitution is an endomorphism of $T(\Sigma, V \cup C)$ with only finitely many variables not mapped to themselves. A substitution is denoted by $\sigma = \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$, where the domain of $\sigma$ is $Dom(\sigma) = \{x_1, \ldots, x_m\}$ and the range of $\sigma$ is $Ran(\sigma) = \{t_1, \ldots, t_m\}$. Application of a substitution $\sigma$ to $t$ is written $t\sigma$. Given a subsignature $\Sigma'$ of $\Sigma$, a $\Sigma'$-*alien subterm* of $t \in T(\Sigma, V \cup C)$ is a $\Sigma \backslash \Sigma'$-rooted subterm of $t$ such that its superterms are $\Sigma'$-rooted. When $\Sigma'$ is clear from the context, a $\Sigma'$-*alien subterm* is called an alien subterm.

**Equational Theories.** Given a set $E$ of $\Sigma$-axioms (i.e., pairs of terms in $T(\Sigma, V)$, denoted by $l = r$), the *equational theory* $=_E$ is the congruence closure of $E$ under the law of substitutivity (by a slight abuse of terminology, $E$ is often called an equational theory). Equivalently, $=_E$ can be defined as the reflexive transitive closure $\leftrightarrow_E^*$ of an equational step $\leftrightarrow_E$ defined as follows: $s \leftrightarrow_E t$ if there exist a position $p$ of $s$, $l = r$ (or $r = l$) in $E$, and substitution $\sigma$ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. An axiom $l = r$ is *regular* if $Var(l) = Var(r)$. An axiom $l = r$ is *collapse-free* if $l$ and $r$ are non-variable terms. An equational theory is *regular* (resp., collapse-free) if all its axioms are regular (resp., collapse-free). An equational theory $E$ is *finite* if for each term $t$, there are only finitely many terms $s$ such that $t =_E s$. A theory $E$ is *syntactic* if it has finite *resolvent presentation* $S$, defined as a finite set of axioms $S$ such that each equality $t =_E u$ has an equational proof $t \leftrightarrow_S^* u$ with at most one equational step $\leftrightarrow_S$ applied at the root position. One can easily check that $C = \{x * y = y * x\}$ (Commutativity) and $AC = \{x * (y * z) = (x * y) * z,\ x * y = y * x\}$ (Associativity-Commutativity) are regular and collapse-free. Moreover, $C$ and $AC$ are syntactic [23]. A $\Sigma$-equation is a pair of $\Sigma$-terms denoted by $s =^? t$ or simply $s = t$ when it is clear from the context that we do not refer to an axiom. A *flat* $\Sigma$-equation is either an equation between variables or a *non-variable flat* $\Sigma$-equation of the form $x_0 = f(x_1, \ldots, x_n)$ where $x_0, x_1, \ldots, x_n$ are variables and $f$ is a function symbol in $\Sigma$. An $E$-unification problem is a set of $\Sigma$-equations, $\Gamma = \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$, or equivalently a conjunction of $\Sigma$-equations. The set of variables in $\Gamma$ is denoted by $Var(\Gamma)$.

A solution to $\Gamma$, called an *E-unifier*, is a substitution $\sigma$ such that $s_i\sigma =_E t_i\sigma$ for all $1 \leq i \leq n$. A substitution $\sigma$ is *more general modulo E* than $\theta$ on a set of variables $V$, denoted as $\sigma \leq_E^V \theta$, if there is a substitution $\tau$ such that $x\sigma\tau =_E x\theta$ for all $x \in V$. $\sigma|_V$ denotes the substitution $\sigma$ restricted to the set of variables $V$. A *Complete Set of E-Unifiers* of $\Gamma$, denoted by $CSU_E(\Gamma)$, is a set of substitutions such that each $\sigma \in CSU_E(\Gamma)$ is an $E$-unifier of $\Gamma$, and for each $E$-unifier $\theta$ of $\Gamma$, there exists $\sigma \in CSU_E(\Gamma)$ such that $\sigma \leq_E^{Var(\Gamma)} \theta$. An *E-unification algorithm* is an algorithm that computes a finite $CSU_E(\Gamma)$ for all $E$-unification problems $\Gamma$. An inference rule $\Gamma \vdash \Gamma'$ for $E$-unification is *sound* if each $E$-unifier of $\Gamma'$ is an $E$-unifier of $\Gamma$; and *complete* if for each $E$-unifier $\sigma$ of $\Gamma$, there exists an $E$-unifier $\sigma'$ of $\Gamma'$ such that $\sigma' \leq_E^{Var(\Gamma)} \sigma$. A set of equations $\Gamma = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$ is said to be in *solved form* if each $x_i$ is a variable occurring once in $\Gamma$. Given an idempotent substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ (such that $\sigma\sigma = \sigma$), $\hat{\sigma}$ denotes the corresponding solved form. An inference system for $E$-unification is *sound* if all its inference rules are sound; and *complete* if for each $E$-unification problem $\Gamma$ on which an inference applies and each $E$-unifier $\sigma$ of $\Gamma$, there exist an $E$-unification problem $\Gamma'$ inferred from $\Gamma$ and an $E$-unifier $\sigma'$ of $\Gamma'$ such that $\sigma' \leq_E^{Var(\Gamma)} \sigma$. To simplify the notation in our inference rules, we apply them modulo the commutativity of $=^?$ and we often use tuples of terms, such as $\bar{u} = (u_1, \ldots, u_n)$, $\bar{v} = (v_1, \ldots, v_n)$ to represent the set of equations $\bar{u} =^? \bar{v}$ corresponding to $\{u_1 =^? v_1, \ldots, u_n =^? v_n\}$.

**Equational Rewrite Relations.** Given a signature $\Sigma$, an oriented $\Sigma$-axiom is called a rewrite rule of the form $l \to r$ such that $l, r \in T(\Sigma, V)$, $l$ is not a variable and $Var(r) \subseteq Var(l)$. Let $R$ be a set of rewrite rules and $E$ an equational $\Sigma$-theory. For any $\Sigma$-terms $s$ and $t$, $s$ *R, E-rewrites* to $t$, denoted by $s \to_{R,E} t$, if there exist a position $p$ of $s$, $l \to r \in R$, and substitution $\sigma$ such that $s_{|p} =_E l\sigma$ and $t = s[r\sigma]_p$. The term $s$ is said to be *R, E-reducible*, $s_{|p}$ is called a *redex*, and in the particular case where $s_{|p} = l\sigma$, $s$ *R-rewrites* to $t$, denoted by $s \to_R t$. A term is an *innermost redex* if none of its proper subterms is a redex. The symmetric relation $\leftarrow_R \cup \to_R \cup =_E$ is denoted by $\longleftrightarrow_{R\cup E}$. The rewrite relation $\to_{R,E}$ is Church-Rosser modulo $E$ if $\longleftrightarrow_{R\cup E}^*$ is included in $\to_{R,E}^* \circ =_E \circ \leftarrow_{R,E}^*$. The rewrite relation $\to_R$ is $E$-terminating if $=_E \circ \to_R \circ =_E$ is terminating. When $\to_R$ is $E$-terminating, $\to_{R,E}$ is Church-Rosser modulo $E$ iff $\to_{R,E}$ is both locally $E$-confluent and locally $E$-coherent [21]. The rewrite relation $\to_{R,E}$ is *E-convergent* if $\to_R$ is $E$-terminating and $\to_{R,E}$ is Church-Rosser modulo $E$. When $\to_{R,E}$ is $E$-convergent, we have that for any terms $t, t'$, $t \longleftrightarrow_{R\cup E}^* t'$ iff $t{\downarrow_{R,E}} =_E t'{\downarrow_{R,E}}$, where $t{\downarrow_{R,E}}$ (resp., $t'{\downarrow_{R,E}}$) denotes any normal form of $t$ (resp., $t'$) w.r.t $\to_{R,E}$. A function symbol that does not occur in $\{l(\epsilon) \mid l \to r \in R\}$ is called a *constructor* for $R$. Let $\Sigma_0$ be the subsignature of $\Sigma$ that consists of function symbols occurring in the axioms of $E$. An $E$-convergent rewrite relation $\to_{R,E}$ is said to be *E-constructed* if all the symbols in $\Sigma_0$ are constructors for $R$. When $R$ is a finite set of rules, the pair $(R, E)$ is called an *equational term rewrite system* (TRS). We say that a property is satisfied by an equational TRS $(R, E)$ if this property is satisfied by $\to_{R,E}$. Given a TRS $(R, E)$, $R^=$ denotes the set of equalities $\{l = r \mid l \to r \in R\}$, and $R^= \cup E$ is the *equational theory of* $(R, E)$. For sake of brevity, we may use $R \cup E$ instead of $R^= \cup E$. The rewrite relation $\to_{R,E}$ and all the related notions introduced above for a set $R$ of rules $l \to r$ such that $l, r \in T(\Sigma, V)$ are extended in a natural way to any set $R$ of ground rules $l \to r$ such that $l, r \in T(\Sigma, C)$ and for which the condition $Var(r) \subseteq Var(l)$ is trivially satisfied since $Var(l) = Var(r) = \emptyset$.

For any equational $\Sigma$-theory $F$, an *F-canonizer stable by renaming* is an idempotent mapping $w : T(\Sigma, V) \to T(\Sigma, V)$ such that for any $s, t \in T(\Sigma, V)$, $s =_F t$ iff $w(s) = w(t)$; for any $t \in T(\Sigma, V)$, $Var(w(t)) \subseteq Var(t)$ and for any variable renaming $\phi$ whose domain is $Var(t)$, $w(t\phi) = w(t)\phi$. For any finite theory $E$ (resp., any $E$-convergent TRS where $E$ is finite), an $E$-canonizer (resp. a $R \cup E$-canonizer) stable by renaming is computable.

## 3 $E$-Constructed Theories and their Combinations

We introduce a class of $E$-constructed theories including $E$-constructed TRSs. In this paper, an $E$-constructed theory $F$ is an equational theory $F$ such that $F \cup E$ admits a particular normalizing mapping over ground terms to compute a normal form for each equivalence class modulo $=_{F \cup E}$. To get an $E$-constructed theory, the normal forms must satisfy some particular properties. In previous papers [14, 13, 18, 19], these properties were expressed using a reduction ordering on ground terms. Here, we adopt the idea of expressing these properties thanks to a normalizing mapping defined as an idempotent mapping on ground terms generated by a countable infinite set $C$ of free constants totally ordered by a well-founded ordering $>$.

▶ **Definition 1** ($>$-compatible renaming)**.** *Assume $C$ is a countable infinite set of constants and $>$ is a well-founded total ordering on $C$, meaning that there is no infinite decreasing sequence $c_1 > c_2 > \dots$ of elements of $C$, and for any $c_1, c_2 \in C$, $c_1 > c_2$ or $c_2 > c_1$ or $c_1 = c_2$. A renaming of a finite subset $Cst$ of $C$ is an injective mapping $\xi$ from $Cst$ to $C$, which is said to be* $>$-compatible *if for any $c_1, c_2 \in Cst$, $c_1 > c_2$ iff $c_1\xi > c_2\xi$. Given a signature $\Sigma$, a renaming $\xi$ of $Cst$ uniquely extends to an endomorphism of $T(\Sigma, C)$, also denoted by $\xi$.*

Through the rest of the paper, we assume that $C$ is a countable infinite set of constants, $>$ is a well-founded total ordering on $C$, $\Sigma_0$ and $\Sigma$ are two signatures such that $\Sigma_0 \subseteq \Sigma$, $E$ is a regular and collapse-free $\Sigma_0$-theory and $F$ is a $\Sigma$-theory.

Let $G$ be a subset of $T(\Sigma, C)$ including $C$. A $\Sigma_0$-term over a set of terms $G$ is a term $u\sigma$ such that $u \in T(\Sigma_0, V)$ and $\sigma$ is a substitution such that $Dom(\sigma) = Var(u)$ and $Ran(\sigma) \subseteq G$. By a slight abuse of notation, the set of $\Sigma_0$-terms over $G$ is denoted by $T(\Sigma_0, G)$. A *constant abstraction mapping modulo $F \cup E$ for $G$* is a mapping $\pi : G\backslash C \to D$ such that $D$ is a set of constants disjoint from $C$ and for any $s, t \in G\backslash C$, $s =_{F \cup E} t$ iff $\pi(s) = \pi(t)$. An inverse mapping of $\pi$ is any morphism $\pi^{-1} : D \to G\backslash C$ such that for any $t \in G\backslash C$, $\pi^{-1}(\pi(t)) =_{F \cup E} t$. For any $t \in T(\Sigma_0, G)$, $t^{\pi_0}$ is called the 0-abstraction of $t$ and is inductively defined as follows:

- $(f(t_1, \dots, t_m))^{\pi_0} = f(t_1^{\pi_0}, \dots, t_m^{\pi_0})$ if $f \in \Sigma_0$,
- $t^{\pi_0} = \pi(t)$ if $t \in G\backslash C$,
- $c^{\pi_0} = c$ if $c \in C$.

Following [6], $G$ is called a $\Sigma_0$-*base* of $F \cup E$ if for any term $t \in T(\Sigma, C)$ there exists a term $s \in T(\Sigma_0, G)$ such that $t =_{F \cup E} s$, and for any $s, s' \in T(\Sigma_0, G)$, $s =_{F \cup E} s'$ iff $s^{\pi_0} =_{F \cup E} s'^{\pi_0}$.

▶ **Definition 2** ($E$-constructed theory)**.** *Let $C$ be a countable infinite set of constants, $>$ a well-founded total ordering on $C$, $\Sigma_0$ and $\Sigma$ two signatures such that $\Sigma_0 \subseteq \Sigma$, $E$ a regular and collapse-free $\Sigma_0$-theory and $F$ a $\Sigma$-theory. An* $E$-constructed normalizing mapping *for $F \cup E$ is an idempotent mapping $NF : T(\Sigma, C) \to T(\Sigma, C)$ with the following properties:*

- *for any $s, t \in T(\Sigma, C)$, $s =_{F \cup E} t$ iff $NF(s) =_E NF(t)$,*
- *for any $t \in T(\Sigma, C)$, $Cst(NF(t)) \subseteq Cst(t) \cup \{c_0\}$, where $c_0$ is minimal in $C$ w.r.t $>$,*
- *for any $t \in T(\Sigma, C)$ and any $>$-compatible renaming $\xi$ of $Cst(t) \cup \{c_0\}$ such that $c_0\xi = c_0$, we have $NF(t\xi) = NF(t)\xi$,*
- *for any $f \in \Sigma_0$, any $t_1, \dots, t_m \in T(\Sigma, C)$, $NF(f(t_1, \dots, t_m)) =_E f(NF(t_1), \dots, NF(t_m))$,*
- *for any $c \in C$, $NF(c) = c$.*
- *Let $G = \{t \mid t \in T(\Sigma, C), t(\epsilon) \in (\Sigma\backslash\Sigma_0) \cup C,$ and $NF(t) = t\}$. For any $t \in T(\Sigma, C)$, $NF(t) \in T(\Sigma_0, G)$.*

*$F$ is said to be $E$-*constructed *if there exists an $E$-constructed normalizing mapping for $F \cup E$. $G$ is called the* $\Sigma_0$-base associated to $NF$*. A term $t \in T(\Sigma, C)$ is* $NF$-normalized *if $NF(t) = t$. A substitution $\sigma$ is* $NF$-normalized *if for each $x \in Dom(\sigma)$, $x\sigma$ is $NF$-normalized.*

In Definition 2, the $\Sigma_0$-base associated to $NF$ is actually a $\Sigma_0$-base of $F \cup E$. Therefore, $\Sigma_0$ is a set of constructors for $F \cup E$, following the definition of constructor studied in [6]. By Definition 2, we have that $=_{F \cup E}$ and $=_E$ coincide on $\Sigma_0$-terms. Thus, for any $E$-constructed theory $F$, $\Sigma_0$ is a set of constructors for $F \cup E$ and the $\Sigma_0$-reduct of $F \cup E$ is $E$. Moreover, note that $F \cup E$-equality is decidable if $NF$ is computable and $E$-equality is decidable.

▶ **Proposition 3.** *For any $E$-constructed TRS $(R, E)$, $R$ is an $E$-constructed theory such that an $E$-constructed normalizing mapping $NF$ for $R \cup E$ is defined as follows: for any $t \in T(\Sigma, C)$, $NF(t) = t{\downarrow_{R,E}}$.*

▶ **Example 4.** Through the rest of the paper we will include several examples using the following axioms: $EX = \{e(e(x, y), z) = e(x, y * z)\}$ for exponentiation, $H = \{e(x * y, z) = e(x, z) * e(y, z)\}$, for the homomorphism like property of exponentiation, $EXH = EX \cup H$, and $AC$ for the $AC$ theory of $*$. For each $F = EX, H, EXH$, the theory $F \cup AC$ is finite, and so the $F \cup AC$-matching problem is finitary. However, the unification problem is undecidable for $EXH \cup AC$ and $H \cup AC$ [27]. For each $F = EX, H, EXH$, orienting the equalities from left to right in $F$ leads to an $AC$-constructed TRS denoted by $(F^{\rightarrow}, AC)$. Then, the $AC$-constructed $(F^{\rightarrow}, AC)$ provides an $AC$-constructed normalized mapping $NF$ since normal forms are stable by variable renaming in equational convergent rewrite systems. Thus, for each $F = EX, H, EXH$, there exists an $AC$-constructed normalizing mapping $NF$ for $F \cup AC = F^{\rightarrow} \cup AC$, meaning that $F$ is $AC$-constructed. For all these $AC$-constructed theories, the $\Sigma_0$-base $G$ associated to $NF$ corresponds to the set of $NF$-normalized terms rooted by a symbol not equal to $*$. Notice, if $NF(t)$ is rooted by $e$ then $NF(t) \in G$ and so $NF(t) \in T(\Sigma_0, G)$. When $t$ is not in $G$, $NF(t)$ is not necessarily in $G$. Consider for instance $F = EX$, $t = e(e(a, b), c)*a$, and $t' = a*e(a, c*b)$. Then $NF(t) = e(a, b*c)*a$ and $NF(t') = t'$ are terms in $T(\Sigma_0, G)\backslash G$. Since $t =_{EX \cup AC} t'$, we have $NF(t) =_{AC} NF(t')$. Assume a constant abstraction mapping $\pi$ modulo $EX \cup AC$ for $G$ such that $\pi(e(a, b * c)) = \pi(e(a, c * b)) = d$ since $e(a, b * c) =_{AC} e(a, c * b)$. One can check that $NF(t)^{\pi_0} = d * a =_{AC} a * d = NF(t')^{\pi_0}$.

▶ **Example 5.** Note, Definition 2 does not require that the theory be orientable into an $E$-constructed TRS. Theories satisfying a commutative property over an $AC$-symbol, $*$, such as $PC = \{fc(x * y, v * w) = fc(v * w, x * y)\}$ and $PCC = PC \cup \{fc(o(x), o(y)) = o(x) * o(y)\}$, satisfy Definition 2. For $F = PC, PCC$, one can check that $F$ is $AC$-constructed and $F \cup AC$ is a finite syntactic theory. For the $AC$-constructed theory $EX$ defined in Example 4, $EX \cup AC$ is also a finite syntactic theory. Actually, the syntacticness of $EX \cup AC$ follows from [23] since $EX \cup AC$ is collapse-free and $EX \cup AC$-unification is known to be finitary [16]. While $F \cup AC$ is not orientable into an $AC$-constructed TRS for $F = PC, PCC$, Example 4 introduces an $AC$-constructed TRS for $EX$.

Unsurprisingly, any $E$-constructed theory corresponds to an $E$-convergent rewrite relation on ground terms. In that case, the corresponding set of rules is infinite and so this rewrite relation cannot be used in practice to compute the normal forms. In an $E$-constructed TRS, the rules are built over terms with variables and they are stable by instantiation. In an $E$-constructed theory, the corresponding rules are ground and a particular notion of stability is considered to allow a renaming of constants, provided that the renaming is $>$-compatible. By Definition 2, a normal form of a term does not depend on the names used to denote the constants, but it depends on the ordering of the constants in the term. Since the equational

theory $F \cup E$ is not necessarily regular in Definition 2, a normal form of a term $t$ may have some additional constants not occurring in $t$. However, a single additional constant suffices, and by Definition 2, it will be the minimal one w.r.t $>$.

▶ **Lemma 6.** *Let $F$ be an $E$-constructed theory, and $NF$ an $E$-constructed normalizing mapping for $F \cup E$. Let $R_{NF}$ be the set of ground rules $t \to NF(t)$ such that $t \in T(\Sigma, C)$, $NF(t) \neq t$, $t(\epsilon) \in \Sigma \backslash \Sigma_0$ and any strict subterm of $t$ is $NF$-normalized. Then, $\to_{R_{NF},E}$ is $E$-constructed and for any $t \in T(\Sigma, C)$, $NF(t) =_E t{\downarrow}_{R_{NF},E}$.*

▶ **Example 7.** Continuing from Example 4, consider any $AC$-constructed TRS $(F^{\to}, AC)$ where $F = EX, H, EXH$. Let $NF$ be the $E$-constructed normalizing mapping such that for any $t \in T(\Sigma, C)$, $NF(t) = t{\downarrow}_{F^{\to},AC}$. By Lemma 6, the normal forms w.r.t $(F^{\to}, AC)$ coincide with the normal forms w.r.t $\to_{R_{NF},AC}$ on $T(\Sigma, C)$.

When $NF$ is an $E$-constructed normalizing mapping for $F \cup E$, a normal form $t{\downarrow}_{R_{NF},E}$ is also simply denoted by $t{\downarrow}_{NF}$. In contrast to [14, 13, 18, 19], the class of $E$-constructed theories given by Definition 2 is closed by non-disjoint union sharing only symbols in $E$. In other words, the class of $E$-constructed theories is modular:

▶ **Theorem 8.** *Assume $F_1$ and $F_2$ are two $E$-constructed theories sharing only symbols in $E$ such that for $i = 1, 2$, $NF_i$ is an $E$-constructed normalizing mapping for $F_i \cup E$. Then, $NF_1$ and $NF_2$ can be extended to an $E$-constructed normalizing mapping $NF_{1,2}$ for $F_1 \cup F_2 \cup E$.*

▶ **Example 9.** Continuing from Examples 5 and 7, for each $F = EX, H, EXH, PC, PCC$ and for each integer $i \geq 1$, let $F_i$ be the theory obtained from $F$ by replacing any function symbol $f$ in $F$ not equal to $*$ by $f_i$. For instance, $F_i = EX_i = \{e_i(e_i(x, y), z) = e_i(x, y * z)\}$ if $F = EX$, and $F_i = H_i = \{e_i(x * y, z) = e_i(x, z) * e_i(y, z)\}$ if $F = H$. Theorem 8 allows us to combine any number of theories $F_i$.

From now on, $F_i$ is assumed to be an $E$-constructed theory with an $E$-constructed normalizing mapping $NF_i$ for the $\Sigma_i$-theory $F_i \cup E$, where $i = 1, 2$. Then, $F_1 \cup F_2 \cup E$ is an $E$-constructed theory. The $E$-constructed normalizing mapping $NF_{1,2}$ derived from $NF_1$ and $NF_2$ by Theorem 8 is simply denoted by $NF$, $G$ is the $\Sigma_0$-base associated to $NF$ corresponding to the set of $((\Sigma_1 \cup \Sigma_2) \backslash \Sigma_0) \cup C$-rooted $NF$-normalized terms, and $\pi$ is a constant abstraction mapping modulo $F_1 \cup F_2 \cup E$ for $G$. Given any $i = 1, 2$ and the subsignature $\Sigma_i$ of $\Sigma_1 \cup \Sigma_2$, a *term with true $i$-aliens* is a term $t$ such that for any $\Sigma_i$-alien subterm $u$ of $t$, $u{\downarrow}_{NF}$ is $\Sigma_{3-i} \backslash \Sigma_0$-rooted. Given any term $t$ with true $i$-aliens, *the $i$-abstraction of $t$* is denoted by $t^{\pi_i}$ and defined as follows:

- for any $f \in \Sigma_i$ and any terms $t_1, \ldots, t_m$, $(f(t_1, \ldots, t_m))^{\pi_i} = f(t_1^{\pi_i}, \ldots, t_m^{\pi_i})$,
- for any $\Sigma_{3-i} \backslash \Sigma_0$-rooted term $t$, $t^{\pi_i} = \pi(t{\downarrow}_{NF})$,
- for any $c \in C$, $c^{\pi_i} = c$.

Given a substitution $\sigma$ such that $x\sigma$ is a term with true $i$-aliens for any $x \in Dom(\sigma)$, we define $\sigma^{\pi_i} = \{x \mapsto (x\sigma)^{\pi_i} \mid x \in Dom(\sigma)\}$.

▶ **Lemma 10.** *For any $i = 1, 2$ and any term $t$ with true $i$-aliens, $t^{\pi_i} =_{F_i \cup E} (t{\downarrow}_{NF})^{\pi_i}$.*

In general, an $E$-constructed normalizing mapping is not computable. However, we show that it is possible to get an approximation, called layer-reduced form, which is useful to decide the equality modulo a union of theories $F_1 \cup F_2 \cup E$ where both $F_1$ and $F_2$ are $E$-constructed.

▶ **Definition 11** (Layer-reduced form). *Let $\Sigma_0$ and $\Sigma$ be two signatures such that $\Sigma_0 \subseteq \Sigma$, $E$ a $\Sigma_0$-theory, and $F$ an $E$-constructed $\Sigma$-theory with an $E$-constructed normalizing mapping $NF$. A* layer-reduced form *is a term in $T(\Sigma, C)$ defined inductively as follows:*

- *$f(t_1,\dots,t_m)$ is in layer-reduced form if $f\in\Sigma_0$ and for each $k\in[1,m]$, $t_k$ is in layer-reduced form,*
- *$t$ is in layer-reduced form if both $t$ and $t{\downarrow}_{NF}$ are $\Sigma\backslash\Sigma_0$-rooted,*
- *$c$ is in layer-reduced form if $c\in C$.*

*Given any term $s\in T(\Sigma,C)$, a* layer-reduced form of $s$ associated to $NF$ modulo $F\cup E$ *is a layer-reduced form $t$ such that $s=_{F\cup E} t$.*

A layer-reduced term mapping returns a layer-reduced form of any input term.

▶ **Definition 12** (Layer-reduced term mapping). *Let $\Sigma_0$ and $\Sigma$ be two signatures such that $\Sigma_0\subseteq\Sigma$, $E$ a $\Sigma_0$-theory, $F$ an $E$-constructed $\Sigma$-theory with an $E$-constructed normalizing mapping $NF$, and $c_0$ the minimal constant in $C$ w.r.t $>$. A* layer-reduced term mapping associated to $NF$ *for $F\cup E$ is an idempotent mapping $(_)\Downarrow : T(\Sigma,C)\to T(\Sigma,C)$ such that:*

- *for any $t\in T(\Sigma,C)$, $t\Downarrow$ is a layer-reduced form of $t$ associated to $NF$ modulo $F\cup E$ such that $Cst(t\Downarrow)\subseteq Cst(t)\cup\{c_0\}$,*
- *for any $t\in T(\Sigma,C)$ and any $>$-compatible renaming $\xi$ of $Cst(t)\cup\{c_0\}$ such that $c_0\xi=c_0$, we have $(t\xi)\Downarrow=(t\Downarrow)\xi$,*
- *for any $f\in\Sigma_0$ and any terms $t_1,\dots,t_m\in T(\Sigma,C)$, $f(t_1,\dots,t_m)\Downarrow=f(t_1\Downarrow,\dots,t_m\Downarrow)$,*
- *for any $c\in C$, $c\Downarrow=c$.*

*A* $\Downarrow$-ordering *is an $F\cup E$-compatible total ordering $>_\Downarrow$ on $T_\Downarrow=\{t\mid t(\epsilon)\in\Sigma\backslash\Sigma_0, t\Downarrow=t\}$ such that for any $t,t'\in T_\Downarrow$ and any $>$-compatible renaming $\xi$ of $Cst(t)\cup Cst(t')\cup\{c_0\}$ with $c_0\xi=c_0$, we have $t>_\Downarrow t'$ iff $t\xi>_\Downarrow t'\xi$.*

In order to decide $F_1\cup F_2\cup E$-equality in a modular way, we show that a computable layer-reduced term mapping $\Downarrow_i$ and a computable $\Downarrow_i$-ordering for $F_i\cup E$, together with a decidable $F_i\cup E$-equality for $i=1,2$ are sufficient.

▶ **Theorem 13.** *Assume $F_1$ and $F_2$ are two $E$-constructed theories sharing only symbols in $E$ such that $F_i\cup E$ has an $E$-constructed normalizing mapping $NF_i$, a computable layer-reduced term mapping $\Downarrow_i$ associated to $NF_i$, a computable $\Downarrow_i$-ordering, and $F_i\cup E$-equality is decidable for any $i=1,2$. Then, $\Downarrow_1$ and $\Downarrow_2$ (resp. , the $\Downarrow_1$-ordering and the $\Downarrow_2$-ordering) can be extended to a computable layer-reduced term mapping $\Downarrow_{1,2}$ associated to $NF_{1,2}$ (resp., a computable $\Downarrow_{1,2}$-ordering) such that for any $i=1,2$ and any term $t$, $t\Downarrow_{1,2}$ is a term with true $i$-aliens, and $F_1\cup F_2\cup E$-equality is decidable.*

At first glance, the computability of $\Downarrow$ and of its related $\Downarrow$-ordering seems difficult to obtain. Fortunately, there is a large class of theories for which we get for free the computability of these mappings and related orderings. The following lemmas are very useful to apply our combination results, e.g., Theorem 13:

▶ **Lemma 14.** *For any $E$-constructed theory $F$ with a computable $F\cup E$-canonizer stable by renaming, any computable layer-reduced term mapping $\Downarrow$ has a computable $\Downarrow$-ordering.*

▶ **Lemma 15.** *For any $E$-constructed theory $F$ such that $F\cup E$ is a regular theory with an $F\cup E$-matching algorithm, a layer-reduced term mapping $\Downarrow$ is computable.*

**Proof.** Consider the procedure defined as the repeated application of the following inference with a don't care non-determinism:

**Expand** $\quad (u,t)\ \vdash\ (u\sigma,t)$
where $x\in Var(u)$, $f\in\Sigma_0$, $\bar{v}$ are fresh variables, $\sigma=\{x\mapsto f(\bar{v})\}$, $CSU_{F\cup E}(\{u\sigma=t\})\neq\emptyset$.

Given any variable $x$, any term $t \in T(\Sigma, C)$ and the input pair $(x, t)$, the above procedure is necessarily terminating since $F$ is $E$-constructed, computing a single pair $(u', t)$ such that $CSU_{F\cup E}(\{u' = t\}) \neq \emptyset$, and for any $\sigma \in CSU_{F\cup E}(\{u' = t\})$, $u'\sigma$ is a layer-reduced form of $t$ modulo $F \cup E$. ◀

▶ Remark 16. The $E$-constructed theory $F_i$ is said to be *$E$-inner* if the normal form by $NF_i$ of any $\Sigma_i\backslash\Sigma_0$-rooted term in $T(\Sigma_i, C)$ remains $\Sigma_i\backslash\Sigma_0$-rooted. When $F_i$ is $E$-inner, the identity mapping provides a layer-reduced term mapping $\Downarrow_i$ for $F_i \cup E$. If both $\Downarrow_1$, $\Downarrow_2$ are the identity mapping, then $\Downarrow_{1,2}$ remains the identity mapping.

When $\Downarrow_i$ is given by a computable $NF_i$ for $i = 1, 2$, $\Downarrow_{1,2}$ corresponds to the computable $NF_{1,2}$. Let us also mention the disjoint case $(\Sigma_0, E) = (\emptyset, \emptyset)$, where $\Downarrow_{1,2}$ is obtained without using an additional computable $\Downarrow_i$-ordering for $i = 1, 2$.

▶ **Example 17.** Continuing from Example 9, we have a computable $NF$ for each theory $F \cup AC$ where $F = EX, H, EXH$. Notice, each of these $NF$s satisfies Definition 12 and provides a layer-reduced term mapping, $\Downarrow$. $EX$ and $H$ are regular theories, thus no new constant $c_0$ is introduced by $\Downarrow$ and $Cst(t\Downarrow) = Cst(t)$. According to Remark 16, $EX$ and $PC$ are $AC$-inner theories for which a layer-reduced term mapping can be provided by the identity mapping. Contrary to $PC$, $PCC$ is not $AC$-inner but $PCC \cup AC$ is finite and we can rely on Lemma 15 to get a computable layer-reduced term mapping. For $H$ and $EXH$, the corresponding computable $NF$ can be used as a layer-reduced term mapping. Thus, we have a computable $\Downarrow$ for each $F \cup AC$ where $F = EX, H, EXH, PC, PCC$. Applying Theorem 13 we obtain, in a modular way, a computable layer-reduced term mapping for $F_1 \cup \cdots F_n \cup AC$, where $F = EX, H, EXH, PC, PCC$. Recall that the construction of the combined layer-reduced term mapping requires computable $\Downarrow$-orderings. For each $F = EX, H, EXH, PC, PCC$, there exists a computable $\Downarrow$-ordering since Lemma 14 applies. Finally, note that Theorem 13 does not require that the component theories $F_i \cup AC$ are regular. In the particular case of a non-regular $AC$-constructed TRS, the layer-reduced term mapping $\Downarrow$ provided by the corresponding computable $NF$ satisfies $Cst(t\Downarrow) \subseteq Cst(t)$ for any term $t$, and Lemma 14 still applies to get a computable $\Downarrow$-ordering since $AC$ is a finite theory.

Consider now the problem of building an $F_1 \cup F_2 \cup E$-matching algorithm where both $F_1$ and $F_2$ are assumed to be regular (as well as $E$). In that case, any matching problem has only ground solutions: given any equation $s =^?_{F_1\cup F_2\cup E} t$ with $t \in T(\Sigma_1 \cup \Sigma_2, C)$ and any substitution $\sigma$ such that $s\sigma =_{F_1\cup F_2\cup E} t$, $\{s\sigma\} \cup Ran(\sigma) \subseteq T(\Sigma_1 \cup \Sigma_2, C)$. The following corollary is a direct consequence of Lemma 10 and paves the way for an $F_1 \cup F_2 \cup E$-matching procedure combining an $F_1 \cup E$-matching algorithm and an $F_2 \cup E$-matching algorithm:

▶ **Corollary 18.** *For any $i = 1, 2$, any $\Sigma_i$-term $s$, any term $t \in T(\Sigma_1 \cup \Sigma_2, C)$ such that $t\Downarrow_{1,2} = t$, and any $NF$-normalized substitution $\sigma$, $s\sigma =_{F_1\cup F_2\cup E} t$ iff $s(\sigma^{\pi_i}) =_{F_i\cup E} t^{\pi_i}$.*

By Corollary 18, the terminating procedure [29, 32, 15, 33] combining the matching algorithms in regular theories remains sound and complete in our extended setting.

▶ **Theorem 19.** *If $F_1$ and $F_2$ are two $E$-constructed theories sharing only symbols in $E$ such that $F_i \cup E$ is a regular theory with a computable layer-reduced term mapping $\Downarrow_i$, a computable $\Downarrow_i$-ordering and an $F_i \cup E$-matching algorithm for $i = 1, 2$, then $F_1 \cup F_2$ is $E$-constructed and $F_1 \cup F_2 \cup E$ is a regular theory with a computable layer-reduced term mapping $\Downarrow_{1,2}$, a computable $\Downarrow_{1,2}$-ordering and an $F_1 \cup F_2 \cup E$-matching algorithm.*

Theorem 19 can be applied to finite theories since any finite theory is a particular case of a regular (and collapse-free) theory with a computable layer-reduced term mapping $\Downarrow$ (cf. Lemma 15), a computable $\Downarrow$-ordering (cf. Lemma 14), and a matching algorithm. Indeed, the matching problem is known to be finitary in any finite theory, thanks to a reduction to syntactic matching via the enumeration of the finitely terms in a given equivalence class modulo the theory. This brute-force method should be avoided whenever it is possible.

## 4 Finite Syntactic Theories and their Combinations

In this section, we focus on the class of finite syntactic theories. In that class, any theory has a mutation-based matching algorithm. The class of finite syntactic theories is known to be closed by disjoint union [28]. More precisely, if $F_1$ and $F_2$ are signature-disjoint finite theories and $F_i$ has a resolvent presentation $S_i$ for $i = 1, 2$, then $F_1 \cup F_2$ is finite and has a resolvent presentation $S_1 \cup S_2$. In the non-disjoint case where $F_1$ and $F_2$ are $E$-constructed theories sharing only symbols in $E$ and $F_i \cup E$ has a resolvent presentation $S_i$ for $i = 1, 2$, it is easy to see that $S_1 \cup S_2$ is not necessarily a resolvent presentation of $F_1 \cup F_2 \cup E$:

▶ **Example 20.** Consider $(\Sigma_0, E) = (\{c\}, \emptyset)$ and $(\Sigma_i, F_i) = (\{f_i, c\}, \{f_i(x) = c(x)\})$ for $i = 1, 2$. $F_i$ is a resolvent presentation of $F_i \cup E$ and $F_i$ is $E$-constructed for $i = 1, 2$ but $F_1 \cup F_2$ cannot be a resolvent presentation of $F_1 \cup F_2 \cup E$ since $f_1(x) =_{F_1 \cup F_2 \cup E} f_2(x)$.

To get that the resolvent presentation of $F_1 \cup F_2 \cup E$ is the union of the resolvent presentations of $F_1 \cup E$ and $F_2 \cup E$, a restricted class of $E$-constructed theories is needed:

▶ **Definition 21** ($E$-capped theory)**.** *Let $E$ be a regular and collapse-free $\Sigma_0$-theory, $F$ an $E$-constructed $\Sigma$-theory with an $E$-constructed normalizing mapping $NF$ for $F \cup E$, and $G$ the $\Sigma_0$-base associated to $NF$. The $E$-constructed normalizing mapping $NF$ is said to be* $E$-capped *if for any $\Sigma \backslash \Sigma_0$-rooted term $t \in T(\Sigma, C)$, $NF(t)$ is a term $u\sigma \in T(\Sigma_0, G)$ such that $u \in T(\Sigma_0, V)$, $Var(u) = Dom(\sigma)$ and $Ran(\sigma) \subseteq G \backslash C$. An $E$-constructed theory $F$ with an $E$-capped normalizing mapping $NF$ is said to be* $E$-capped.

▶ **Example 22.** In Definition 21, the term $u$ can be a variable and so any $E$-inner theory as defined in Remark 16 is $E$-capped. Consider the theories defined in Examples 4 and 5. For $F = EX, PC$, the theory $F$ is $E$-capped since $F$ is $E$-inner. For $F = H, EXH, PCC$, the theory $F$ is $E$-capped without being $E$-inner.

When $F_1$ and $F_2$ are two $E$-capped theories sharing only symbols in $E$, for any $\Sigma_1 \backslash \Sigma_0$-rooted term $t_1$ and any $\Sigma_2 \backslash \Sigma_0$-rooted term $t_2$, $t_1$ cannot be equal to $t_2$ modulo $F_1 \cup F_2 \cup E$. In [19], the following result has been shown: if $F_1$ and $F_2$ are two $E$-capped theories sharing only symbols in $E$ and $F_i \cup E$ is regular collapse-free with a resolvent presentation $S_i$ for $i = 1, 2$, then $F_1 \cup F_2$ is $E$-capped and $F_1 \cup F_2 \cup E$ is regular collapse-free with a resolvent presentation $S_1 \cup S_2$.

▶ **Example 23.** Consider $(\Sigma_0, E) = (\{c\}, \emptyset)$ and $(\Sigma_i, F_i) = (\{f_i, g_i, c\}, \{f_i(x) = c(g_i(x))\})$ for $i = 1, 2$. $F_i$ is a resolvent presentation of $F_i \cup E$ and $F_i$ is $E$-capped and regular collapse-free for $i = 1, 2$. By the modularity result in [19] mentioned above, $F_1 \cup F_2$ is $E$-capped and $F_1 \cup F_2$ is a resolvent presentation of $F_1 \cup F_2 \cup E$.

When $F_i \cup E$ is finite, only finitely many distinct non-normalized terms can have the same normal form w.r.t $NF_{1,2}$. Since, for any $s, t$, $s =_{F_1 \cup F_2 \cup E} t$ iff $NF_{1,2}(s) =_E NF_{1,2}(t)$ where $E$ is necessarily finite, we have that $F_1 \cup F_2 \cup E$ is finite too.

▶ **Theorem 24.** *If $F_1$ and $F_2$ are two $E$-capped theories sharing only symbols in $E$ such that $F_i \cup E$ is a finite theory with a resolvent presentation $S_i$ for $i = 1, 2$, then $F_1 \cup F_2$ is $E$-capped and $F_1 \cup F_2 \cup E$ is a finite theory with a resolvent presentation $S_1 \cup S_2$.*

With $E$-constructed TRSs, another resolvence allows us to get rid of the $E$-capped assumption.

▶ **Definition 25** (Innermost-resolvent $E$-constructed TRS)**.** *An $E$-constructed TRS $(R, E)$ is said to be* innermost-resolvent *if any innermost rewrite derivation $s \rightarrow^*_{R,E} t$ includes at most one rewrite step applied at the root position. An innermost-resolvent TRS $(R, E)$ is* finite *if $R \cup E$ is finite.*

▶ **Example 26.** Continuing from Example 7, consider any $AC$-constructed TRS $(F^{\rightarrow}, AC)$ where $F = EX, H, EXH$. Applying the rule corresponding to $EX$ more than once at the root would violate the innermost strategy. The rule corresponding to $H$ moves the constructor symbol $*$ to the root and thus disallows any further root rewriting. Thus, $(F^{\rightarrow}, AC)$ is innermost-resolvent for each $F = EX, H, EXH$.

Following the terminology in [10], $R \cup E$ is 2-syntactic when $(R, E)$ is innermost-resolvent.

▶ **Theorem 27.** *Let $(R_1, E)$ and $(R_2, E)$ be two finite innermost-resolvent $E$-constructed TRSs sharing only symbols in $E$. If $\rightarrow_{R_1 \cup R_2}$ is $E$-terminating, then $(R_1 \cup R_2, E)$ is a finite innermost-resolvent $E$-constructed TRS.*

▶ **Example 28.** Continuing from Examples 20 and 23, consider $(\Sigma_0, E) = (\{c\}, \emptyset)$, $(\Sigma_1, R_1) = (\{f_1, c\}, \{f_1(x) \rightarrow c(x)\})$ and $(\Sigma_2, R_2) = (\{f_2, g_2, c\}, \{f_2(x) \rightarrow c(g_2(x))\})$. $(R_1, E)$ and $(R_2, E)$ are two finite innermost-resolvent $E$-constructed TRSs sharing only symbols in $E$ and $\rightarrow_{R_1 \cup R_2}$ is $E$-terminating. By Theorem 27, $(R_1 \cup R_2, E)$ is a finite innermost-resolvent $E$-constructed TRS.

## 5 Hierarchical Matching

**Norm** $\{s = t\} \cup \Gamma \ \vdash \ \{s = t\Downarrow\} \cup \Gamma$
where $s$ is a non-ground term and $t$ is a ground term such that $t\Downarrow \neq t$.

**Triv** $\{s = t\} \cup \Gamma \ \vdash \ \Gamma$
where $s, t$ are ground terms such that $s\Downarrow = s$, $t\Downarrow = t$, and $s =_{F \cup E} t$.

**Figure 1** $NT$ rules.

We investigate in this section the problem of building an $F \cup E$-matching algorithm in the case $F$ is $E$-constructed, $F \cup E$ has a computable layer-reduced term mapping $\Downarrow$, a computable $\Downarrow$-ordering, and an $E$-matching algorithm is known. A hierarchical matching algorithm for $F \cup E$ is defined as an inference system including the inference rules in $NT \cup HM_E$ where $NT$ and $HM_E$ are respectively given in Figure 1 and in Figure 2. The rules in $NT$ are clearly sound and complete in $F \cup E$, by definition of $\Downarrow$. The rules in $HM_E \backslash \{$**Solve-M**$\}$ are sound and complete in any equational theory. To show that **Solve-M** is sound and complete in $F \cup E$, we rely on the 0-abstraction of a term in layer-reduced form. The 0-abstraction of any term $t \in T(\Sigma_0, G)$, denoted by $t^{\pi_0}$, has been introduced just before Definition 2 and it can be extended to a larger set of terms. A *term with true* 0*-aliens* is a term $t$ such that for any $\Sigma_0$-alien subterm $u$ of $t$, $u{\downarrow}_{NF}$ is $\Sigma \backslash \Sigma_0$-rooted. Given any term $t$ with true 0-aliens, *the* 0*-abstraction of* $t$ is denoted by $t^{\pi_0}$ and defined as follows:

**Rep** $\{x = t\} \cup \Gamma \ \vdash \ \{x = t\} \cup (\Gamma\{x \mapsto t\})$
where $x$ is a variable occurring in $\Gamma$ and $t$ is a ground term.

**Flatten-M** $\{f(\bar{u}) = t\} \cup \Gamma \ \vdash \ \{f(\bar{x}) = t, \bar{u} = \bar{x}\} \cup \Gamma$
where $f(\bar{u})$ is a non-ground $\Sigma \backslash \Sigma_0$-rooted term, $t$ is ground, and $\bar{x}$ are fresh variables.

**VA-M** $\{s[u] = t\} \cup \Gamma \ \vdash \ \{s[x] = t, u = x\} \cup \Gamma$
where $s$ is a non-ground $\Sigma_0$-rooted term, $u$ is a $\Sigma_0$-alien subterm of $s$, $t$ is a ground, and $x$ is a fresh variable.

**Solve-M** $\Gamma \cup \Gamma_0 \ \vdash \ \Gamma \cup \hat{\sigma}$
where $\Gamma_0 = \{s_k = t_k\}_{k \in K}$, $s_k \in T(\Sigma_0, V \cup C)$ and $t_k \in T(\Sigma, C)$ for each $k \in K$, $\Gamma_0^{\pi_0} = \{s_k = t_k^{\pi_0}\}_{k \in K}$, $CSU_E(\Gamma_0^{\pi_0}) \neq \emptyset$, $\sigma_0 \in CSU_E(\Gamma_0^{\pi_0})$, and $\hat{\sigma}$ is the solved form of $\sigma = \sigma_0 \pi^{-1}$.

**Figure 2** $HM_E$ rules.

- for any $f \in \Sigma_0$ and any terms $t_1, \dots, t_m$, $(f(t_1, \dots, t_m))^{\pi_0} = f(t_1^{\pi_0}, \dots, t_m^{\pi_0})$,
- for any $\Sigma \backslash \Sigma_0$-rooted term $t$, $t^{\pi_0} = \pi(t{\downarrow}_{NF})$,
- for any $c \in C$, $c^{\pi_0} = c$.

Given a substitution $\sigma$ such that $x\sigma$ is a term with true 0-aliens for any $x \in Dom(\sigma)$, we define $\sigma^{\pi_0} = \{x \mapsto (x\sigma)^{\pi_0} \mid x \in Dom(\sigma)\}$.

▶ **Lemma 29.** *For any term $t$ with true 0-aliens, $t^{\pi_0} =_E (t{\downarrow}_{NF})^{\pi_0}$.*

▶ **Lemma 30.** *For any ground terms $s, t$ in layer-reduced form, we have:*
- *if $s, t$ are $\Sigma_0$-rooted or $s, t \in C$, then $s =_{F \cup E} t \Leftrightarrow s^{\pi_0} =_E t^{\pi_0}$,*
- *if $s$ is $\Sigma_0$-rooted and $t$ is $\Sigma \backslash \Sigma_0$-rooted or $s \in C$ and $t$ is $\Sigma$-rooted, then $s \neq_{F \cup E} t$.*

▶ **Corollary 31.** *For any ground term $t$ in layer-reduced form, any $\Sigma_0$-term $s$ and any $NF$-normalized substitution $\sigma$, $s\sigma =_{F \cup E} t$ iff $s(\sigma^{\pi_0}) =_E t^{\pi_0}$.*

Corollary 31 follows from Lemma 29. It shows that **Solve-M** is sound and complete in $F \cup E$. To solve any $F \cup E$-matching problem, we need to complete $NT \cup HM_E$ by some inference system, say $U$, to transform the match-equations that cannot be handled by $NT \cup HM_E$.

▶ **Definition 32** (Hierarchical matching algorithm)**.** *Assume an $E$-matching algorithm, a computable layer-reduced term mapping $\Downarrow$ for $F \cup E$, and an inference system $U$ satisfying the following assumptions:*

**(a)** *no single inference rule in $U$ is sound and complete for an arbitrary equational theory;*
**(b)** *$U$ is sound and complete for $F \cup E$ provided that all the inference rules in $U$ are applied using a don't know non-determinism;*
**(c)** *each equation that can be solved by **Solve-M** must remain unchanged by $U$.*

*A* hierarchical matching algorithm for $F \cup E$ *is an inference system denoted by $HM_E(\Downarrow, U)$ and defined by the set of rules in $NT \cup HM_E \cup U$ (cf. Figures 1 and 2) such that the following properties hold for any input set $\Gamma$ of equations $s = t$ where $s$ or $t$ is ground:*
- *the repeated application of rules in $HM_E(\Downarrow, U)$ terminates with the following order of priority: **Norm**, **Triv**, **Rep**, **Flatten-M**, **VA-M**, $U$, **Solve-M**;*
- *any normal form of $\Gamma$ w.r.t $HM_E(\Downarrow, U)$ obtained by the above strategy is $F \cup E$-unifiable iff it is a matching problem in solved form.*

By definition, any hierarchical matching algorithm for $F \cup E$ is a sound and complete $F \cup E$-matching algorithm. In the following, we give examples of theories with hierarchical matching algorithms.

▶ **Lemma 33.** *Let $DM_R$ be the inference system given in Figure 3. For any finite innermost-resolvent $E$-constructed TRSs $(R, E)$, $R \cup E$ admits a hierarchical matching algorithm of the form $HM_E(\downarrow_{R,E}, DM_R)$.*

In Lemma 33, the soundness and completeness of $DM_R$ follows directly from the assumption that $(R, E)$ is innermost-resolvent and the fact that ground terms are normalized before any rule from $DM_R$ applies. Since $R \cup E$ is finite, $HM_E(\downarrow_{R,E}, DM_R)$ is terminating.

| | | | | |
|---|---|---|---|---|
| **Dec** | $\{f(\bar{v}) = f(\bar{t})\} \cup \Gamma$ | $\vdash$ | $\{\bar{v} = \bar{t}\} \cup \Gamma$ | where $f \in \Sigma \backslash \Sigma_0$ |
| **Mut**$_R$ | $\{f(\bar{v}) = g(\bar{t})\} \cup \Gamma$ | $\vdash$ | $\{\bar{v} = \bar{l}, \bar{r} = \bar{t}\} \cup \Gamma$ | where $f(\bar{l}) \to g(\bar{r}) \in R$ |

**Figure 3** $DM_R$ rules.

▶ **Example 34.** Continuing from Example 26, for each $F = EX, H, EXH$, the $AC$-constructed TRS $(F^{\to}, AC)$ is innermost-resolvent and $F^{\to} \cup AC$ has a hierarchical matching algorithm of the form $HM_{AC}(\downarrow_{F^{\to},AC}, DM_{F^{\to}})$.

▶ **Lemma 35.** *Assume $F$ is $E$-constructed and $F \cup E$ is a finite theory with a resolvent presentation $S$ and a computable layer-reduced term mapping $\Downarrow$. Let $DM_S$ be the inference system obtained from the one in Figure 3 by replacing any rule from $R$ by an equality from $S$. Then $F \cup E$ has a hierarchical matching algorithm of the form $HM_E(\Downarrow, DM_S)$.*

In Lemma 35, the soundness and completeness of $DM_S$ follows directly from the assumption that $S$ is a resolvent presentation of $F \cup E$. In addition, $HM_E(\Downarrow, DM_S)$ is terminating since $F \cup E$ is finite.

▶ **Example 36.** For $F = EX, PC, PCC$, we have that $F \cup AC$ is a finite theory with a resolvent presentation $S$ and a computable layer-reduced term mapping $\Downarrow$. Thus, $HM_{AC}(\Downarrow, DM_S)$ is a hierarchical matching algorithm for $F \cup AC$. The resolvent presentation $S$ includes some $\Sigma_0$-equalities for $\Sigma_0 = \{*\}$. These $\Sigma_0$-equalities, corresponding to a resolvent presentation of $AC$, are not used in the application of $DM_S$.

## 6 Hierarchical Matching in Combined $E$-Constructed Theories

In our hierarchical approach, combining hierarchical matching algorithms parameterized by $U_1$ and $U_2$ can be viewed as a hierarchical matching algorithm parameterized by $U_1 \cup U_2$. The following remark details how the inference rules in $U_i$ involving ground $\Sigma_i$-terms are extended to handle ground $\Sigma_1 \cup \Sigma_2$-terms.

▶ Remark 37. Assume an $F_i \cup E$-matching algorithm of the form $HM_E(\Downarrow_i, U_i)$ for $i = 1, 2$. The inference system $U_i$ is defined for matching-equations with ground terms in $T(\Sigma_i, C)$. To handle ground terms in $T(\Sigma_1 \cup \Sigma_2, C)$, $U_i$ must be extended in the expected manner via $i$-abstraction, leading to a signature extension of $U_i$ defined as follows for any problem $P$ including some function symbol in $\Sigma_{3-i} \backslash \Sigma_0$: $P \vdash_{U_i} Q\pi^{-1}$ if $P^{\pi_i} \vdash_{U_i} Q$, where $P^{\pi_i}$ denotes the $E_i$-matching problem obtained from $P$ by replacing each ground side $t$ in $P$ by $t^{\pi_i}$. This is sound and complete by Corollary 18, and the fact that ground sides in $P$ are in layer-reduced form since **Norm** is applied eagerly before $U_i$. In the same way, **Solve-M** has been extended to handle ground sides in $T(\Sigma_1 \cup \Sigma_2, C)$. In that case, the 0-abstraction is used to get an $E$-matching problem.

▶ **Theorem 38.** *If $F_1$ and $F_2$ are two $E$-constructed theories sharing only symbols in $E$ such that $F_i \cup E$ is a regular theory with a computable layer-reduced term mapping $\Downarrow_i$, a computable $\Downarrow_i$-ordering, and a hierarchical matching algorithm of the form $HM_E(\Downarrow_i, U_i)$ for $i = 1, 2$. Then $F_1 \cup F_2$ is an $E$-constructed and $F_1 \cup F_2 \cup E$ is a regular theory with a computable layer-reduced term mapping $\Downarrow_{1,2}$, a computable $\Downarrow_{1,2}$-ordering, and a hierarchical matching algorithm of the form $HM_E(\Downarrow_{1,2}, U_1 \cup U_2)$.*

**Proof.** The combination algorithm for the matching problem that allows us to obtain Theorem 19 can be expressed as a hierarchical matching algorithm for $F_1 \cup F_2 \cup E$ of the form $HM_E(\Downarrow_{1,2}, \{\textbf{Solve-M}_1, \textbf{Solve-M}_2\})$, where $\textbf{Solve-M}_i$ for $i = 1, 2$ is defined as follows in a way similar to **Solve-M**:

$\textbf{Solve-M}_i \quad \Gamma \cup \Gamma_i \ \vdash \ \Gamma \cup \hat{\sigma}$
where $\Gamma_i = \{s_k = t_k\}_{k \in K}$, $s_k \in T(\Sigma_i \backslash \Sigma_0, V \cup C)$, $t_k \in T(\Sigma_1 \cup \Sigma_2, C)$ for each $k \in K$, $\Gamma_i^{\pi_i} = \{s_k = t_k^{\pi_i}\}_{k \in K}$, $CSU_{F_i \cup E}(\Gamma_i^{\pi_i}) \neq \emptyset$, $\sigma_i \in CSU_{F_i \cup E}(\Gamma_i^{\pi_i})$, and $\hat{\sigma}$ is the solved form of $\sigma = \sigma_i \pi^{-1}$.

Assume $F_i \cup E$ has a hierarchical matching algorithm of the form $HM_E(\Downarrow_i, U_i)$ for any $i = 1, 2$. Then, $\textbf{Solve-M}_i$ can be replaced by $HM_E(\Downarrow_i, U_i)$. Due to the rule application strategy used in any hierarchical matching algorithm, $\textbf{Solve-M}_i$ applies only on match-equations $s = t$ such that $s$ is a flat non-ground $\Sigma_i \backslash \Sigma_0$-term, and $t$ is a ground term in layer-reduced form w.r.t $\Downarrow_{1,2}$. Thus, $U_i$ is sufficient to replace $\textbf{Solve-M}_i$ for $i = 1, 2$, and so the combination matching algorithm is actually of the form $HM_E(\Downarrow_{1,2}, U_1 \cup U_2)$. ◀

▶ **Example 39.** In Examples 34 and 36, we have shown that $F_i \cup AC$ has a hierarchical matching algorithm for each $F = EX, H, EXH, PC, PCC$. By Theorem 38, $F_1 \cup \cdots \cup F_n \cup AC$ has a hierarchical matching algorithm.

Notice, Theorem 38 applies to $E$-constructed regular theories $F_i$ where $F_i \cup E$ is not necessarily finite. In the particular case of finite theories, we get the following corollaries.

▶ **Corollary 40.** *Assume $(R_1, E)$ and $(R_2, E)$ are two finite innermost-resolvent $E$-constructed TRSs sharing only symbols in $E$. If $\to_{R_1 \cup R_2}$ is $E$-terminating, then $(R_1 \cup R_2, E)$ is a finite innermost-resolvent $E$-constructed TRS and $R_1 \cup R_2 \cup E$ admits a hierarchical matching algorithm of the form $HM_E(\downarrow_{R_1 \cup R_2, E}, DM_{R_1} \cup DM_{R_2})$.*

Corollary 40 is a continuation of Theorem 27. Interestingly, the hierarchical matching algorithm for $R_1 \cup R_2 \cup E$ can be obtained from Theorem 38 but also as a consequence of Lemma 33 since $HM_E(\downarrow_{R_1 \cup R_2, E}, DM_{R_1 \cup R_2})$ coincides with $HM_E(\downarrow_{R_1 \cup R_2, E}, DM_{R_1} \cup DM_{R_2})$.

▶ **Example 41.** Continuing from Examples 9 and 34, we can combine any number of exponentiation/homomorphic theories $F_i \cup AC$ for $F = EX, H, EXH$ sharing only the $AC$-symbol $*$ and obtain a hierarchical matching algorithm of the form given by Corollary 40.

▶ **Corollary 42.** *If $F_1$ and $F_2$ are two $E$-capped theories sharing only symbols in $E$ such that $F_i \cup E$ is a finite theory with a resolvent presentation $S_i$, a computable layer-reduced term mapping $\Downarrow_i$, and a hierarchical matching algorithm of the form $HM_E(\Downarrow_i, DM_{S_i})$ for $i = 1, 2$. Then $F_1 \cup F_2$ is $E$-capped and $F_1 \cup F_2 \cup E$ is a finite theory with a resolvent presentation $S_1 \cup S_2$, a computable layer-reduced term mapping $\Downarrow_{1,2}$, and a hierarchical matching algorithm of the form $HM_E(\Downarrow_{1,2}, DM_{S_1} \cup DM_{S_2})$.*

Corollary 42 is a continuation of Theorem 24. Again, the hierarchical matching algorithm for $F_1 \cup F_2 \cup E$ can be obtained from Theorem 38 but also as a consequence of Lemma 35 since $HM_E(\Downarrow_{1,2}, DM_{S_1 \cup S_2})$ coincides with $HM_E(\Downarrow_{1,2}, DM_{S_1} \cup DM_{S_2})$.

▶ **Example 43.** Continuing from Examples 9 and 36, we can combine any number of finite syntactic theories $F_i \cup AC$ for $F = EX, PC, PCC$ sharing only the $AC$-symbol $*$ and obtain a finite syntactic theory with a hierarchical matching algorithm of the form given by Corollary 42.

## 7 Hierarchical Decision Procedures for the Word-Problem

In a hierarchical matching algorithm for $F \cup E$, it is mandatory to be able to decide $F \cup E$-equality of terms in layer-reduced form (cf. rules in $NT$, Figure 1). In a way similar to hierarchical $F \cup E$-matching, it is possible to follow a simple hierarchical approach for solving the particular $F \cup E$-unification problem where both sides of each equation are ground terms in layer-reduced form. In that particular case, we assume a decidable $E$-equality, an $E$-constructed theory $F$, and a computable layer-reduced term mapping $\Downarrow$ for $F \cup E$. Consider the following inference rule:

$$\textbf{Solve-W} \quad \{s = t\} \cup \Gamma \vdash \Gamma \quad \text{where } (s(\epsilon), t(\epsilon) \in \Sigma_0 \text{ or } s, t \in C) \text{ and } s^{\pi_0} =_E t^{\pi_0}$$

together with an inference system $U$ satisfying the same assumptions (a) and (b) as in Definition 32 and for which each equation that can be solved by **Solve-W** must remain unchanged by $U$. A *hierarchical decision procedure for the $F \cup E$-equality of terms in layer-reduced form w.r.t $\Downarrow$* is an inference system denoted by $HW_E(U)$ and defined by the set of rules in $\{\textbf{Solve-W}\} \cup U$ such that, for any input set $\Gamma$ of equations $s = t$ where $s$ and $t$ are ground terms in layer-reduced form w.r.t $\Downarrow$, the repeated application of rules in $HW_E(U)$ terminates with the order of priority $U, \textbf{Solve-W}$, and the empty set of equations is the unique $F \cup E$-unifiable normal form w.r.t $HW_E(U)$. By definition, $HW_E(U)$ is a sound, complete, and terminating procedure deciding the $F \cup E$-equality of terms in layer-reduced form. There are two major classes of $E$-constructed theories $F$ with a hierarchical decision procedure for the $F \cup E$-equality of terms in layer-reduced form:

1. If $(R, E)$ is an $E$-constructed TRS and $E$ is finite, then $HW_E(\{\textbf{Dec}\})$ is a hierarchical decision procedure for the $R \cup E$-equality of terms in layer-reduced form w.r.t $\downarrow_{R,E}$, where **Dec** is given in Figure 3. This holds since the $\Sigma \backslash \Sigma_0$-symbols do not occur in $E$.
2. If $F$ is $E$-constructed and $F \cup E$ is a finite theory with a resolvent presentation $S$, then $HW_E(\{\textbf{Dec}, \textbf{Mut-W}_S\})$ is a hierarchical decision procedure for the $F \cup E$-equality of terms in layer-reduced form, where **Dec** is given in Figure 3 and $\textbf{Mut-W}_S$ is as follows:
   $\textbf{Mut-W}_S \quad \{f(\bar{v}) = g(\bar{t})\} \cup \Gamma \vdash \Gamma \quad \text{where } f(\bar{l}) = g(\bar{r}) \in S,\ CSU_{F \cup E}(\{\bar{l} = \bar{v}, \bar{r} = \bar{t}\}) \neq \emptyset.$
   This can be shown using the same proof argument as in Lemma 35.

The class of $E$-constructed theories $F$ with a hierarchical decision procedure for the $F \cup E$-equality of terms in layer-reduced form satisfies a modular property described below.

▶ **Theorem 44.** *Under the same assumptions as in Theorem 13, if $HW_E(U_i)$ is a hierarchical decision procedure for the $F_i \cup E$-equality of terms in layer-reduced form w.r.t $\Downarrow_i$, for $i = 1, 2$, then, $HW_E(U_1 \cup U_2)$ is a hierarchical decision procedure for the $F_1 \cup F_2 \cup E$-equality of terms in layer-reduced form w.r.t $\Downarrow_{1,2}$.*

**Proof.** By Theorem 13, $HW_E(\{\textbf{Solve-W}_1, \textbf{Solve-W}_2\})$ is a hierarchical decision procedure for the $F_1 \cup F_2 \cup E$-equality of terms in layer-reduced form w.r.t $\Downarrow_{1,2}$, where for $i = 1, 2$, $\textbf{Solve-W}_i$ is as follows:

$$\textbf{Solve-W}_i \quad \{s = t\} \cup \Gamma \vdash \Gamma \quad \text{where } s(\epsilon), t(\epsilon) \in \Sigma_i \backslash \Sigma_0 \text{ and } s^{\pi_i} =_{F_i \cup E} t^{\pi_i}.$$

The use of $U_i$ being extended to ground mixed terms via $i$-abstraction (cf. Remark 37), **Solve-W**$_i$ can be replaced by $U_i$. ◀

Notice, Theorem 44 applies to $E$-constructed theories $F_i$ where $F_i \cup E$ can be non-regular.

## 8 Related Work and Concluding Remarks

The theories $F \cup E$ we are interested in are conservative extensions of $E$ for which the symbols in the signature $\Sigma_0$ of $E$ are constructors, meaning that $F \cup E$ admits a $\Sigma_0$-basis [6, 35]. In [6], a modularity result was shown for the computability of normal forms over the $\Sigma_0$-basis. This result requires that normal forms are stable by variable renaming. In contrast, we rely on a stability by constant renaming, provided that the renaming follows an arbitrary total ordering over the constants. Moreover, we give a modular construction for computable layer-reduced term mappings which are sufficient approximations of the normalizing mappings used to define the $E$-constructed theories. The notion of layer-reduced form is well-known in the context of disjoint combination [31], but this is the first time a modular construction of layer-reduced forms is proposed for theories sharing constructors modulo $E$. The combination problem for both unification and matching in constructor-sharing theories has been investigated for a while [12, 5, 32, 14, 15, 33] but we now consider the general case of constructors modulo $E$ to go beyond the case of absolutely free constructors. We have shown that our hierarchical approach is a well-suited framework to deal with non-absolutely free constructors. This hierarchical approach has been initiated to study the unification problem in various classes of $E$-constructed theories [18, 19]. As shown here, the restriction to the matching problem allows us to get hierarchical matching algorithms for larger classes of $E$-constructed theories, and this completes the terminating cases that have been recently identified for hierarchical unification [18, 19]. The modularity results shown here for the matching problem can be viewed as non-disjoint extensions of the ones known in the disjoint case for the matching problem both in regular theories [29] and in finite syntactic theories [28].

In the future, we are interested in developing new decision procedures for combined theories sharing constructors modulo $E$. More precisely, we target the knowledge problems considered in protocol analysis, for which some first results have been obtained for combined theories sharing absolutely free constructors [17]. Again, the hierarchical approach seems very useful to move from absolutely free constructors to constructors modulo $E$. More generally, our project consists in applying the hierarchical approach to constraint solving problems that occur in protocol analysis, including particular forms of disunification problems.

## References

1. Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.
2. Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
3. Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symb. Comput.*, 21(2):211–243, 1996.
4. Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001.
5. Franz Baader and Cesare Tinelli. Combining decision procedures for positive theories sharing constructors. In Sophie Tison, editor, *Rewriting Techniques and Applications, 13th Interna-*

*tional Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2002.

**6** Franz Baader and Cesare Tinelli. Deciding the word problem in the union of equational theories. *Inf. Comput.*, 178(2):346–390, 2002.

**7** Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.

**8** Bruno Blanchet. Modeling and verifying security protocols with the Applied Pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.

**9** Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theor. Comput. Sci.*, 285(2):155–185, 2002.

**10** Alexandre Boudet and Evelyne Contejean. On $n$-syntactic equational theories. In Hélène Kirchner and Giorgio Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2-4, 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 446–457. Springer, 1992.

**11** Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

**12** Eric Domenjoud, Francis Klay, and Christophe Ringeissen. Combination techniques for non-disjoint equational theories. In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 1994.

**13** Ajay Kumar Eeralla, Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Rule-based unification in combined theories and the finite variant property. In Carlos Martín-Vide, Alexander Okhotin, and Dana Shapira, editors, *Language and Automata Theory and Applications - 13th International Conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings*, volume 11417 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2019.

**14** Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Paliath Narendran, and Christophe Ringeissen. Hierarchical combination. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2013.

**15** Serdar Erbatur, Deepak Kapur, Andrew M. Marshall, Paliath Narendran, and Christophe Ringeissen. Unification and matching in hierarchical combinations of syntactic theories. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 291–306. Springer, 2015.

**16** Serdar Erbatur, Andrew M. Marshall, Deepak Kapur, and Paliath Narendran. Unification over distributive exponentiation (sub)theories. *J. Autom. Lang. Comb.*, 16(2-4):109–140, 2011.

**17** Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Notions of knowledge in combinations of theories sharing constructors. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2017.

**18** Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Terminating non-disjoint combined unification. In Maribel Fernández, editor, *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*, volume 12561 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2020.

**19** Serdar Erbatur, Andrew M. Marshall, and Christophe Ringeissen. Non-disjoint combined unification and closure by equational paramodulation. In Boris Konev and Giles Reger, editors,

*Frontiers of Combining Systems - 13th International Symposium, FroCoS 2021, Birmingham, UK, September 8-10, 2021, Proceedings*, volume 12941 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2021.

**20** Santiago Escobar, Catherine A. Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design, Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2007.

**21** Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.

**22** Deepak Kapur, Paliath Narendran, and Lida Wang. An E-unification algorithm for analyzing protocols that use modular exponentiation. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 165–179. Springer, 2003.

**23** Claude Kirchner and Francis Klay. Syntactic theories and unification. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 270–277. IEEE Computer Society, 1990.

**24** Catherine Meadows and Paliath Narendran. A unification algorithm for the group Diffie-Hellman protocol. In *Informal Proceedings of the Workshop on Issues in the Theory of Security (WITS)*, 2002.

**25** Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

**26** Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2003.

**27** Paliath Narendran. Solving linear equations over polynomial semirings. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 466–472. IEEE Computer Society, 1996.

**28** Tobias Nipkow. Proof transformations for equational theories. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 278–288. IEEE Computer Society, 1990.

**29** Tobias Nipkow. Combining matching algorithms: The regular case. *J. Symb. Comput.*, 12(6):633–654, 1991.

**30** Christophe Ringeissen. Unification in a combination of equational theories with shared constants and its application to primal algebras. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 261–272. Springer, 1992.

**31** Christophe Ringeissen. Combining decision algorithms for matching in the union of disjoint equational theories. *Inf. Comput.*, 126(2):144–160, 1996.

**32** Christophe Ringeissen. Matching in a class of combined non-disjoint theories. In Franz Baader, editor, *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2003.

**33** Christophe Ringeissen. Building and combining matching algorithms. In Carsten Lutz, Uli Sattler, Cesare Tinelli, Anni-Yasmin Turhan, and Frank Wolter, editors, *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His*

*60th Birthday*, volume 11560 of *Lecture Notes in Computer Science*, pages 523–541. Springer, 2019.

**34** Manfred Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symb. Comput.*, 8(1/2):51–99, 1989.

**35** Cesare Tinelli and Christophe Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theor. Comput. Sci.*, 290(1):291–353, 2003.

# A Technical Appendix

Let us first introduce an ordering on $T(\Sigma, C)$ that will be useful in our proofs. This ordering reuses the classical LPO reduction ordering [2] which is defined with respect to a precedence. Actually, any total ordering on $T(\Sigma, C)$ would work provided that it is stable by $>$-compatible renaming. Assume an arbitrary total ordering $>_{\Sigma\cup C}$ on $\Sigma \cup C$ such that the restriction of $>_{\Sigma\cup C}$ to $C$ is $>$, and all the symbols in $\Sigma$ are greater than all the symbols in $C$ w.r.t $>_{\Sigma\cup C}$. For any $s, t \in T(\Sigma, C)$, we write $s >_{LPO} t$ if $s$ is greater than $t$ w.r.t the LPO ordering whose precedence is given by the restriction of $>_{\Sigma\cup C}$ to $\Sigma \cup Cst(s) \cup Cst(t)$.

In a straightforward way, any $F$-canonizer stable by renaming corresponds to an idempotent mapping from $T(\Sigma, C)$ to $T(\Sigma, C)$, also denoted by $w$, such that for any $s, t \in T(\Sigma, C)$, $s =_F t$ iff $w(s) = w(t)$; for any $t \in T(\Sigma, C)$, $Cst(w(t)) \subseteq Cst(t)$ and for any $>$-compatible renaming $\xi$ of $Cst(t) \cup \{c_0\}$ with $c_0\xi = c_0$, $w(t\xi) = w(t)\xi$.

▶ **Definition 45.** *Let $F$ be an equational $\Sigma$-theory, and $w$ an $F$-canonizer stable by renaming. Given any terms $t, t' \in T(\Sigma, C)$, we define $t >_w t'$ if $w(t) >_{LPO} w(t')$.*

▶ **Lemma 46.** *The ordering $>_w$ given in Definition 45 satisfies the following properties:*
- *$>_w$ is $F$-compatible.*
- *For any $t, t' \in T(\Sigma, C)$, we have that either $t =_F t'$ or $t >_w t'$ or $t' >_w t$.*
- *For any $t, t' \in T(\Sigma, C)$ and any $>$-compatible renaming $\xi$ of $Cst(t) \cup Cst(t') \cup \{c_0\}$ with $c_0\xi = c_0$, we have $t\xi >_w t'\xi$ iff $t >_w t'$.*

**Proof.** Consider any $u, t, t', u' \in T(\Sigma, C)$.
- $u =_F t >_w t' =_F u'$ implies $w(u) = w(t) >_{LPO} w(t') = w(u')$, and so $u >_w u'$.
- Since $>_{LPO}$ is total, we have $t >_w t'$ or $t' >_w t$ for any $t, t'$ such that $t \neq_F t'$.
- For any $>$-compatible renaming $\xi$ of $Cst(t) \cup Cst(t') \cup \{c_0\}$ with $c_0\xi = c_0$, we have $t\xi >_w t'\xi$ iff $w(t\xi) >_{LPO} w(t'\xi)$ iff $w(t)\xi >_{LPO} w(t')\xi$. Due to the chosen precedence for the LPO ordering, we have $w(t)\xi >_{LPO} w(t')\xi$ iff $w(t) >_{LPO} w(t')$. Thus, $t\xi >_w t'\xi$ iff $t >_w t'$. ◀

## A.1 Theorems

In the next two proofs, we use an additional notion of constant abstraction mapping:

▶ **Definition 47.** *Assume $F$ is an equational $\Sigma$-theory, $Cst$ is a finite subset of $C$, $AT$ is a finite subset of $T(\Sigma, C)\backslash C$ such that $(\bigcup_{u\in AT} Cst(u)) \subseteq Cst$, $NC$ is a finite subset of $C\backslash(Cst \cup \{c_0\})$, and $\gg$ is an $F$-compatible ordering which is total on $AT$. A mapping $\Pi : AT \to NC$ is said to be a* $(\gg, =_F)$-ordered constant abstraction mapping with a range out of $Cst$ *if for any $u, v \in AT$, $\Pi(u) > \Pi(v)$ iff $u \gg v$ and $\Pi(u) = \Pi(v)$ iff $u =_F v$. Under these assumptions, $\Pi^{-1}$ is any arbitrary morphism from $NC$ to $AT$ such that for any $u \in AT$, $(\Pi(u))\Pi^{-1} =_F u$. For any term $t \in T(\Sigma, C)\backslash C$ such that $Cst(t) \subseteq Cst$, $t^\Pi$ denotes the term obtained from $t$ by replacing any subterm $u$ of $t$ occurring in $AT$ by $\Pi(u)$.*

For any $\Sigma_i\backslash\Sigma_0$-rooted term $t$, the set of $\Sigma_i$-alien subterms of $t$ is denoted by $Alien(t)$.

**Proof of Theorem 8.** Given two $E$-constructed normalizing mappings $NF_1$ and $NF_2$ for $F_1 \cup E$ and $F_2 \cup E$ respectively, we show how to combine them in order to construct an $E$-constructed normalizing mappings $NF_{1,2}$ for $F_1 \cup F_2 \cup E$ in a way $NF_{1,2}$ coincides with $NF_i$ on $T(\Sigma_i, C)$ for any $i = 1, 2$.

Consider $w$ is any $E$-canonizer stable by renaming. Since $w$ does not need to be computable, such a mapping always exists. Then, $>_w$ is the ordering given in Definition 45.

$NF_{1,2}$ is inductively defined as follows:

- For any $c \in C$, $NF_{1,2}(c) = c$.
- Let $t$ be any $\Sigma_0$-rooted term of the form $f(t_1, \dots, t_m)$. Then, we define $NF_{1,2}(t) = f(NF_{1,2}(t_1), \dots, NF_{1,2}(t_m))$.
- Let $t$ be any $\Sigma_i \backslash \Sigma_0$-rooted term. If $Alien(t) = \emptyset$, then $NF_{1,2}(t) = NF_i(t)$. Otherwise, let $t'$ be the term obtained from $t$ by replacing each $u \in Alien(t)$ by $NF_{1,2}(u)$. If $Alien(t') = \emptyset$, then $NF_{1,2}(t) = NF_i(t')$. Otherwise, let $\Pi : Alien(t') \to NC$ be a $(>_w, =_E)$-ordered constant abstraction mapping with a range out of $Cst(t')$. We define $NF_{1,2}(t) = (NF_i(t'^{\Pi}))\Pi^{-1}$.

One can check that $NF_{1,2}$ inherits all the properties stating that $NF_1$ and $NF_2$ are $E$-constructed normalizing mappings, including the property that $NF$ is stable by $>$-compatible renaming (third item of Definition 2) thanks to Lemma 46. ◀

**Proof of Theorem 13.** Let $NF$ be the $E$-constructed normalizing mapping obtained from $NF_1$ and $NF_2$ by applying Theorem 8. Just like any layer-reduced term mapping, it is sufficient to define $\Downarrow_{1,2}$ on $(\Sigma_1 \cup \Sigma_2) \backslash \Sigma_0$-rooted terms. Then, $\Downarrow_{1,2}$ uniquely extends to $\Sigma_0 \cup C$-rooted terms. The definition of $\Downarrow_{1,2}$ bears similarities with the construction of $NF$ detailed in the proof of Theorem 8.

For any $\Sigma_i \backslash \Sigma_0$-rooted term $t$, $t\Downarrow_{1,2}$ is inductively defined as follows:

If $Alien(t) = \emptyset$, then $t\Downarrow_{1,2} = t\Downarrow_i$. Otherwise, let $t'$ the term obtained from $t$ by replacing each $u \in Alien(t)$ by $u\Downarrow_{1,2}$. If $Alien(t') = \emptyset$, then $t\Downarrow_{1,2} = t'\Downarrow_i$. Otherwise, let $\Pi : Alien(t') \to NC$ be a $(>_{\Downarrow_{1,2}}, =_{F_1 \cup F_2 \cup E})$-ordered constant abstraction mapping with a range out of $Cst(t')$. We define $t\Downarrow_{1,2} = ((t'^{\Pi})\Downarrow_i)\Pi^{-1}$ if $(t'^{\Pi})\Downarrow_i \neq t'^{\Pi}$, otherwise $t\Downarrow_{1,2} = t'$.

The $>_{\Downarrow_{1,2}}$ ordering used above is inductively defined as follows:

- Let $s, t$ be any $\Sigma_i \backslash \Sigma_0$-rooted terms such that $s\Downarrow_{1,2} = s$ and $t\Downarrow_{1,2} = t$. If $Alien(s) = Alien(t) = \emptyset$, then $s >_{\Downarrow_{1,2}} t$ iff $s >_{\Downarrow_i} t$. Otherwise, let $\Pi : Alien(s) \cup Alien(t) \to NC$ be a $(>_{\Downarrow_{1,2}}, =_{F_1 \cup F_2 \cup E})$-ordered constant abstraction mapping with a range out of $Cst(s) \cup Cst(t)$. We define $s >_{\Downarrow_{1,2}} t$ iff $s^{\Pi} >_{\Downarrow_i} t^{\Pi}$.
- Let $s$ be any $\Sigma_2 \backslash \Sigma_0$-rooted term such that $s\Downarrow_{1,2} = s$ and $t$ any $\Sigma_2 \backslash \Sigma_0$-rooted term such that $t\Downarrow_{1,2} = t$, we define $s >_{\Downarrow_{1,2}} t$ (this choice is arbitrary).

According to our assumptions on the stability by renaming of both $\Downarrow_i$ and $>_{\Downarrow_i}$, it is important to note that $\Downarrow_{1,2}$ and $>_{\Downarrow_{1,2}}$ are well-defined since we get the same results independently from the chosen $\Pi$. Then, we can prove the following statements:

- For any $\Sigma_i \backslash \Sigma_0$-rooted term $t \in T(\Sigma_1 \cup \Sigma_2, C)$ such that $t\Downarrow_{1,2} = t$, $t$ is a term with true $i$-aliens, $t\downarrow_{NF}$ is $\Sigma_i \backslash \Sigma_0$-rooted, and a renaming of $t^{\pi_i}$ can be effectively built.
- For any $t \in T(\Sigma_1 \cup \Sigma_2, C)$, $t\Downarrow_{1,2}$ is a computable layer-reduced form of $t$ associated to $NF$ modulo $F_1 \cup F_2 \cup E$.

These statements are proved by induction using the height of layers of a term $t \in T(\Sigma_1 \cup \Sigma_2, C)$, denoted by $hl(t)$ and defined as follows:

- If $t$ is a $\Sigma_0$-rooted term $f(t_1, \dots, t_m)$, then $hl(t) = \max_{k=1,\dots,m} hl(t_k)$.
- If $t$ is $\Sigma_i \backslash \Sigma_0$-rooted, then (if $Alien(t) \neq \emptyset$, then $hl(t) = 1 + \max_{u \in Alien(t)} hl(u)$, else $hl(t) = 0$).
- If $t \in C$, then $hl(t) = 0$.

Eventually, the decidability of $F_1 \cup F_2 \cup E$-equality is a direct consequence of Lemma 48 (cf. Section A.2). ◀

**Proof of Theorem 27.** First of all, note that an $E$-convergent TRS $(R, E)$ over the signature $\Sigma$ is innermost-resolvent iff for any $s \in T(\Sigma, V)$, any innermost derivation $s \rightarrow^*_{R,E} s{\downarrow}_{R,E}$ includes at most one rewrite step applied at the root position. This holds because any innermost derivation $s \rightarrow^*_{R,E} t$ can be extended to an innermost derivation $s \rightarrow^*_{R,E} t \rightarrow^*_{R,E} s{\downarrow}_{R,E}$.

Let us now check that $(R_1 \cup R_2, E)$ is $E$-convergent. First, $\rightarrow_{R_1 \cup R_2}$ is assumed to be $E$-terminating. Second, $(R_1 \cup R_2, E)$ is Church-Rosser modulo $E$ since both $(R_1, E)$ and $(R_2, E)$ are $E$-constructed. Consequently, $(R_1 \cup R_2, E)$ is $E$-convergent.

Consider an innermost derivation $s \rightarrow^*_{R_1 \cup R_2, E} s{\downarrow}_{R_1 \cup R_2, E}$, where $s$ is assumed to be $\Sigma_i \backslash \Sigma_0$-rooted for any $i = 1, 2$. This innermost derivation can be divided in two parts. First, we normalize all the alien subterms of $s$, leading to a term $t$ whose aliens are now normalized. Second, we normalize $t$ until $s{\downarrow}_{R_1 \cup R_2, E}$ is reached. Thus, we have an innermost derivation $s \rightarrow^*_{R_1 \cup R_2, E} t \rightarrow^*_{R_1 \cup R_2, E} s{\downarrow}_{R_1 \cup R_2, E}$. All the rules in the innermost derivation $t \rightarrow^*_{R_1 \cup R_2, E} s{\downarrow}_{R_1 \cup R_2, E}$ are necessarily rules from $R_i$ because $t$ is a $\Sigma_i$-rooted term whose alien subterms are normalized, and so the alien subterms remain in the substitution part of any rule application. Consequently, $t \rightarrow^*_{R_i, E} s{\downarrow}_{R_1 \cup R_2, E}$. Since $(R_i, E)$ is innermost-resolvent, $t \rightarrow^*_{R_i, E} s{\downarrow}_{R_1 \cup R_2, E}$ includes at most one rewrite step applied at the root position. Moreover, all the rules in $s \rightarrow^*_{R_1 \cup R_2, E} t$ are applied below the root position. Consequently, $s \rightarrow^*_{R_1 \cup R_2, E} t \rightarrow^*_{R_i, E} s{\downarrow}_{R_1 \cup R_2, E}$ includes at most one rewrite step applied at the root position, and so $(R_1 \cup R_2, E)$ is innermost-resolvent. ◀

## A.2 Lemmas

**Proof of Lemma 6.** Let $\rightarrow_{R_{NF}/E}$ be $=_E \circ \rightarrow_{R_{NF}} \circ =_E$. By definition of $R_{NF}$, $\rightarrow_{R_{NF}/E}$ is an optimally reducing rewrite relation where the length of any derivation starting from any $t \in T(\Sigma, C)$ is bounded by the size of $t$, and so $\rightarrow_{R_{NF}/E}$ is terminating. For any $t \in T(\Sigma, C)$, $t \rightarrow^*_{R_{NF}/E} NF(t)$. Let us check that $=_{F \cup E}$ coincides with $\longleftrightarrow^*_{R_{NF} \cup E}$. For any $s, t \in T(\Sigma, C)$, $s =_{F \cup E} t$ implies $NF(s) =_E NF(t)$ where $s \rightarrow^*_{R_{NF}/E} NF(s)$ and $t \rightarrow^*_{R_{NF}/E} NF(t)$. Thus, $=_{F \cup E} \subseteq \longleftrightarrow^*_{R_{NF} \cup E}$. Conversely, $\longleftrightarrow^*_{R_{NF} \cup E} \subseteq =_{F \cup E}$ since for any $l \rightarrow r \in R_{NF}$, $l =_{F \cup E} r$. For any peak $s \leftarrow_{R_{NF}, E} t \rightarrow_{R_{NF}, E} s'$, we have $s \rightarrow^*_{R_{NF}/E} NF(s) =_E NF(s') \leftarrow^*_{R_{NF}/E} s'$. Since $\Sigma_0$ is a set of constructors for $R_{NF}$, $\rightarrow_{R_{NF}, E}$ is $E$-coherent, and the proof $s \rightarrow^*_{R_{NF}/E} \circ =_E \circ \leftarrow^*_{R_{NF}/E} s'$ can be turned into $s \rightarrow^*_{R_{NF}, E} \circ =_E \circ \leftarrow^*_{R_{NF}, E} s'$. Thus, $\rightarrow_{R_{NF}, E}$ is locally $E$-confluent. Thanks to [21], $\rightarrow_{R_{NF}, E}$ is $E$-convergent, and more precisely $E$-constructed. ◀

**Proof of Lemma 10.** If $t$ is $\Sigma_{3-i} \backslash \Sigma_0$-rooted, then $t^{\pi_i} = \pi(t{\downarrow}_{NF}) = \pi((t{\downarrow}_{NF}){\downarrow}_{NF}) = (t{\downarrow}_{NF})^{\pi_i}$.

For any $\Sigma_i$-rooted $t$, let $t'$ be the term obtained from $t$ by replacing each $\Sigma_i$-alien subterm $u$ of $t$ by $u{\downarrow}_{NF}$. To prove that $(t')^{\pi_i} =_{F_i \cup E} (t{\downarrow}_{NF})^{\pi_i}$ follows from the construction of $NF_{1,2}$, consider $R = R_{NF_{1,2}}$ and $R_i = \{l \rightarrow r \mid l(\epsilon) \in \Sigma_i \backslash \Sigma_0, l \rightarrow r \in R\}$ for $i = 1, 2$. We have the following property: if $s$ is a $\Sigma_i$-rooted term such that its $\Sigma_i$-alien subterms are $NF_{1,2}$-normalized, then $s \rightarrow_{R,E} s'$ implies $s \rightarrow_{R_i,E} s'$, $s'$ is either $NF_{1,2}$-normalized or a $\Sigma_i$-rooted term such that its $\Sigma_i$-alien subterms are $NF_{1,2}$-normalized, and in both cases $s^{\pi_i} =_{F_i \cup E} (s')^{\pi_i}$, by definition of $R_i$. Then, by induction on the length of the derivation $t' \rightarrow^*_{R,E} t{\downarrow}_{NF}$ we prove that $(t')^{\pi_i} =_{F_i \cup E} (t{\downarrow}_{NF})^{\pi_i}$. Finally, we get $t^{\pi_i} =_{F_i \cup E} (t{\downarrow}_{NF})^{\pi_i}$ since $t^{\pi_i} = (t')^{\pi_i}$. ◀

**Proof of Lemma 14.** Let $w$ be a computable $F \cup E$-canonizer stable by renaming, and $>_w$ the corresponding ordering introduced in Definition 45. Given any terms $t, t' \in T_{\Downarrow}$, we define $t >_{\Downarrow} t'$ if $t >_w t'$. Thus, $>_{\Downarrow}$ is computable. By Lemma 46, $>_{\Downarrow}$ fulfills all the properties of a $\Downarrow$-ordering as given in Definition 12. ◀

**Proof of Lemma 29.** If $t$ is $\Sigma\backslash\Sigma_0$-rooted, then $t^{\pi_0} = \pi(t{\downarrow}_{NF}) = \pi((t{\downarrow}_{NF}){\downarrow}_{NF}) = (t{\downarrow}_{NF})^{\pi_0}$. Assume now $t$ is $\Sigma_0$-rooted. Let $t'$ be the term obtained from $t$ by replacing each $\Sigma_0$-alien subterm $u$ of $t$ by $u{\downarrow}_{NF}$. According to Definition 2, we have $(t')^{\pi_0} =_E (t{\downarrow}_{NF})^{\pi_0}$. Since $t^{\pi_0} = (t')^{\pi_0}$, we get $t^{\pi_0} =_E (t{\downarrow}_{NF})^{\pi_0}$. ◀

**Proof of Lemma 30.**

- Consider $s$ and $t$ are $\Sigma_0$-rooted. Then, $s{\downarrow}_{NF}$ and $t{\downarrow}_{NF}$ are $\Sigma_0$-rooted, and we have the following equivalences. First, $s =_{F\cup E} t$ iff $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$. Second, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ iff $(s{\downarrow}_{NF})^{\pi_0} =_E (t{\downarrow}_{NF})^{\pi_0}$ by Definition 2. Then, $(s{\downarrow}_{NF})^{\pi_0} =_E (t{\downarrow}_{NF})^{\pi_0}$ implies that $s^{\pi_0} =_{F\cup E} t^{\pi_0}$ by Lemma 29. Thus, $s^{\pi_0} =_{F\cup E} t^{\pi_0}$ if $s =_{F\cup E} t$. Conversely, by definition of the 0-abstraction, $s =_{F\cup E} t$ if $s^{\pi_0} =_{F\cup E} t^{\pi_0}$.
- Consider $s, t \in C$. Then, $s =_{F\cup E} t$ iff $s^{\pi_0} = s =_E t = t^{\pi_0}$.
- Consider $s$ is $\Sigma_0$-rooted and $t$ is $\Sigma\backslash\Sigma_0$-rooted. Assume $s =_{F\cup E} t$. Then, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ where $s{\downarrow}_{NF}$ is $\Sigma_0$-rooted and $t{\downarrow}_{NF}$ is $\Sigma\backslash\Sigma_0$-rooted. This is impossible since $E$ is regular collapse-free and the symbols in $\Sigma\backslash\Sigma_0$ do not occur in $E$.
- Consider $s \in C$ and $t$ is $\Sigma$-rooted. Assume $s =_{F\cup E} t$. Then, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ where $s{\downarrow}_{NF} \in C$ and $t{\downarrow}_{NF}$ is $\Sigma$-rooted. This is impossible since $E$ is regular collapse-free and any constant in $C$ can only be $E$-equal to itself. ◀

The following lemma is similar to Lemma 30. It is used in the proof of Theorem 13.

▶ **Lemma 48.** *The layer-reduced term mapping $\Downarrow_{1,2}$ associated to $NF_{1,2}$ satisfies the following properties for any $((\Sigma_1 \cup \Sigma_2)\backslash\Sigma_0) \cup C$-rooted terms $s$ and $t$ such that $s\Downarrow_{1,2} = s$ and $t\Downarrow_{1,2} = t$:*

- *if $s, t$ are $\Sigma_i\backslash\Sigma_0$-rooted for some $i = 1, 2$, then $s =_{F_1\cup F_2\cup E} t \Leftrightarrow s^{\pi_i} =_{F_i\cup E} t^{\pi_i}$,*
- *if $s, t \in C$, then $s =_{F_1\cup F_2\cup E} t \Leftrightarrow s = t$,*
- *otherwise, $s \neq_{F_1\cup F_2\cup E} t$.*

**Proof.**

- Consider $s$ and $t$ are $\Sigma_i\backslash\Sigma_0$-rooted for some $i = 1, 2$. Then, $s{\downarrow}_{NF}$ and $t{\downarrow}_{NF}$ are $\Sigma_i\backslash\Sigma_0$-rooted, and we have the following equivalences. First, $s =_{F_1\cup F_2\cup E} t$ iff $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$. Second, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ iff $(s{\downarrow}_{NF})^{\pi_i} =_E (t{\downarrow}_{NF})^{\pi_i}$ since $s{\downarrow}_{NF}$ and $t{\downarrow}_{NF}$ are $\Sigma_i\backslash\Sigma_0$-rooted, and the symbols in $\Sigma_i\backslash\Sigma_0$ do not occur in $E$. Then, $(s{\downarrow}_{NF})^{\pi_i} =_E (t{\downarrow}_{NF})^{\pi_i}$ implies that $s^{\pi_i} =_{F_i\cup E} t^{\pi_i}$ by Lemma 10. Thus, $s^{\pi_i} =_{F_i\cup E} t^{\pi_i}$ if $s =_{F_1\cup F_2\cup E} t$. Conversely, by definition of the $i$-abstraction, $s =_{F_1\cup F_2\cup E} t$ if $s^{\pi_i} =_{F_i\cup E} t^{\pi_i}$.
- Consider $s, t \in C$. Then, $s =_{F_1\cup F_2\cup E} t$ iff $s =_E t$ iff $s = t$.
- Consider $s$ is $\Sigma_1\backslash\Sigma_0$-rooted and $t$ is $\Sigma_2\backslash\Sigma_0$-rooted. Assume $s =_{F_1\cup F_2\cup E} t$. Then, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ where $s{\downarrow}_{NF}$ is $\Sigma_1\backslash\Sigma_0$-rooted and $t{\downarrow}_{NF}$ is $\Sigma_2\backslash\Sigma_0$-rooted. This is impossible since $E$ is regular collapse-free and the symbols in $(\Sigma_1 \cup \Sigma_2)\backslash\Sigma_0$ do not occur in $E$.
- Consider $s \in C$ and $t$ is $\Sigma_1 \cup \Sigma_2$-rooted. Assume $s =_{F_1\cup F_2\cup E} t$. Then, $s{\downarrow}_{NF} =_E t{\downarrow}_{NF}$ where $s{\downarrow}_{NF} \in C$ and $t{\downarrow}_{NF}$ is $\Sigma_1 \cup \Sigma_2$-rooted. This is impossible since $E$ is regular collapse-free and any constant in $C$ can only be $E$-equal to itself. ◀

# Nominal Anti-Unification with Atom-Variables

**Manfred Schmidt-Schauß**
Goethe Universität, Frankfurt am Main, Germany

**Daniele Nantes-Sobrinho**
Department of Computing, Imperial College London, UK
Department of Mathematics, University of Brasília, Brazil

## Abstract

Anti-unification is the task of generalizing a set of expressions in the most specific way. It was extended to the nominal framework by Baumgarter, Kutsia, Levy and Villaret, who defined an algorithm solving the nominal anti-unification problem, which runs in polynomial time. Unfortunately, when an infinite set of atoms are allowed in generalizations, a minimal complete set of solutions in nominal anti-unification does not exist, in general. In this paper, we present a more general approach to nominal anti-unification that uses *atom-variables* instead of explicit atoms, and two variants of freshness constraints: $\mathtt{NL}_A$-constraints (with atom-variables), and EQR-constraints based on EQuivalence Relations on atom-variables. The idea of atom-variables is that different atom-variables may be instantiated with identical or different atoms. Albeit simple, this freedom in the formulation increases its application potential: we provide an algorithm that is finitary for the $\mathtt{NL}_A$-freshness constraints, and for EQR-freshness constraints it computes a unique least general generalization. There is a price to pay in the general case: checking freshness constraints and other related logical questions will require exponential time. The setting of Baumgartner et al. is improved by the atom-only case, which runs in polynomial time and computes a unique least general generalization.

**2012 ACM Subject Classification** Theory of computation → Automated reasoning

**Keywords and phrases** Generalization, anti-unification, nominal algorithms, higher-order deduction

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.7

**Funding** *Daniele Nantes-Sobrinho*: partially funded by the EPSRC Fellowship 'VeTSpec: Verified Trustworthy Software Specification' (EP/R034567/1) and Edital DPI/DPG n. 03/2020.

## 1 Introduction

Anti-unification is the task of, given a set of expressions, to find a most specific (or least general) generalization of all the expressions in this set. In the first-order version, anti-unification simply looks for the largest common term structure and also takes care of equal variables. In this case, the problem can be solved in polynomial time and produces a unique solution [1]. A simple example, taken from Plotkin [21], is the expression $P(g(x), x)$ that generalizes the set $\{P(g(a), a), P(g(b), b)\}$. Notice that the expressions $z$, $P(y, x)$ and $P(g(y), x)$ also generalize the expressions in the set, however, they are not least general: there exist substitutions $\sigma_i$, such that $P(y, x)\sigma_1 = P(g(x), x)$, $P(g(y), x)\sigma_2 = P(g(x), x)$ and $z\sigma_3 = P(g(x), x)$, but not vice versa.

In this paper we are interested in a more complex variation of this problem in the context of a nominal language [11], which is a convenient alternative for expressing languages with binders with the benefit that nominal unification is decidable in quadratic time and unitary [25, 6, 18]. Thus, this work develops around the *nominal anti-unification problem*, i.e., the problem of finding a least general generalization of nominal expressions. Similarly to the relation between nominal unification and higher-order pattern unification [17], nominal anti-unification relates with higher-order pattern anti-unification, thus, developments in nominal

© Manfred Schmidt-Schaußand Daniele Nantes-Sobrinho;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 7; pp. 7:1–7:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

anti-unification promote new insights into the tractability and applicability of higher-order anti-unification problems, and consequently, it has a high potential for applications such as recursion scheme and clone detection [2, 3], learning with counter examples [16], etc.

The nominal anti-unification problem considered by Baumgartner et al. in [4] does not have a least general generalization (lgg), not even a minimal complete set of generalizers can be computed (i.e. it is nullary). A simple example illustrates the infinite set of generalizations: for $f(a_1)$ and $g(a_2)$, the generalization $(\emptyset, X)$ is appropriate, where the pair consists of an empty set of freshness constraints and the generalization variable $X$. However, there is a strictly decreasing chain $(\emptyset, X)$, $(\{a_3\#X\}, X)$,$(\{a_3\#X, a_4\#X\}, X), \ldots$ (where $a_i\#X$ means that atom $a_i$ is fresh in the instances of $X$). These are more and more strictly specific generalizations, for an infinite set of atoms $\{a_3, a_4, \ldots\}$. The names $a_3, a_4, \ldots$ are irrelevant for the problem, but provide an argument that a *least* general generalization might not exist, not even a minimal complete set of lggs. Restricting the set of available atoms to a finite set as in [4] results in nice properties of the algorithm. Although it may suffice in practice, it is not satisfactory.

Our approach is to employ *atom-variables* [24] in the grammar of nominal expressions, which are intended to represent atoms, and formulate the anti-unification problem in this extended nominal language. Atoms are only used in the semantics, whereas in the expression language, only atom-variables are used. Therefore, basic nominal syntactic notions, such as permutations, suspended variables and abstractions, are now generalized to contain both atom-variables $(A, B, C, \ldots)$ and generalization variables $(X, Y, \ldots)$. For instance, abstractions such as $\lambda C.f(C, X)$ and $\lambda(A\ C) \cdot B.f(C, X)$, suspended atom and generalization variables, such as $(A\ B) \cdot C$ and $(A\ B) \cdot X$, are allowed, and their meaning relies on the instantiation of such atom-variables by concrete atoms: mapping atom-variables $A, B, C$ to concrete different atoms $a, b, c$, respectively, $(A\ C) \cdot B$ reduces to $(a\ c) \cdot b = b$ (the permutation $(a\ c)$ has no effect on $b$) and the abstraction $\lambda(A\ C) \cdot B.f(C, X)$ becomes $\lambda b.f(c, X)$. Another important feature is that alpha-equivalence $(\sim)$ is defined semantically: deciding whether $(A\ B) \cdot C \sim (A\ C) \cdot B$ relies on the instantiation of the atoms $A, B$ and $C$. In the case where $B$ and $C$ are mapped to the same atom, the equivalence holds, but it fails if $A, B, C$ are mapped to different atoms.

Standard freshness constraints in our language, called $\mathtt{NL}_A$-constraints, include not only constraints of the form $A\#X$, but also of the forms $A\#B$, $B\#\lambda A.B$ etc. As expected, the meaning of these constraints depends on the instantiation of the atom-variables occurring in them. For instance, $A\#X$ means that instances of $A$ cannot occur free in instances of $X$, and $A\#B$ means that instances of $A$ are different from the instances of $B$. The expressivity gain is nicely observed with $C\#\lambda A.\lambda B.C$, which means that the instantiation of $C$ is the same as that of $A$ or of $B$. However, this is still not sufficient: For representing generalizations in a more expressive way we introduce the Eqr-constraints, which permit, for every equivalence class of instantiations, an extra set of constraints. For example, $((A = B) \implies A\#X) \wedge ((A \neq B) \implies B\#X)$ with the informal meaning: if $A, B$ are equally instantiated, then $A\#X$ must be satisfied, otherwise $B\#X$. This increase permits to have a smaller number of least general generalizations.

Moreover, the standard notion of a nominal expression being *more general* than another (which is relevant for generalizations) when atom-variables are involved is defined by a more complex requirement than just one being an instance of the other: one has to consider the instances of the atom-variables involved in both terms. Thus, the syntax of the terms is more powerful, which means that the syntax of the generalizations is more powerful, and this has to be taken into account by the search for a most specific generalization. This means

that these "extended" freshness constraints and permutations with atom-variables, which are now part of the syntax of nominal terms, also may appear in generalizations, which increases the representative power of this approach.

**Contributions.** We construct a sound and complete rule-based algorithm AtomAntiUnif for computing generalizations (see Section 3), which performs in exponential time, and relies on the subalgorithm EqvM for equivariant matching with atom-variables. Our rules are inspired by [4], modified with semantic checks for equalities and disequalities of atom-variables, and extra rules dealing with suspended atom/generalization variables. Even though our algorithm AtomAntiUnif computes a good generalization, it is not least general, as it is shown in Example 3.4. Our approach significantly improves previous ones: there are no infinite chains of generalizations (Corollary 3.12). Thus, the unification type of generalization problems with atom-variables is finitary.

$\mathtt{NL}_A$- freshness constraints should be investigated with care: a more expressive form of freshness constraints based on EQuivalence Relations over atom-variables has to be considered. The EqR-freshness constraints (Definition 4.2) are a more powerful version of freshness constraints that explicitly represent all possible in-/equality patterns of instantiations of atom-variables. To obtain a singleton solution set, a second algorithm AtomAntiUnifLgg refines the constraint part of the generalization computed by AtomAntiUnif, and returns a least general generalization (Definition 4.10). Finally, we conclude that the anti-unification problem with atom-variables and EqR-freshness constraints is decidable and unitary (Theorem 4.13). With the restriction that different atom-variables can only be instantiated with different atoms (called the atoms-only case), we obtain a specialization of our algorithm that computes a singleton complete set of lggs in polynomial time for our variant of the nominal anti-unification problem investigated in [4] (see Theorem 4.16). This corresponds to the problem in [4] with infinitely many atoms, but using a more flexible semantics. The power of our semantics is illustrated in Example 2.14.

**Related Work.** Early works on generalization date back to the 70s, a well-known note is presented by Plotkin [21], who discussed the usefulness of generalization when looking for methods of induction. More recent works such as [7] and [8] investigate higher-order pattern anti-unification, the latter is an extension with equational theories. Applications of anti-unification were exploited in several directions, such as finding parallel recursion schemes [2], in program analysis [5], for analogy (or clone) detection [3, 14, 19], in description logics [13], checking inductive properties of term rewriting systems [10], among others. A library of anti-unification algorithms is available in [3].

**Organization.** Section 2 gives the necessary background on the nominal language $\mathtt{NL}_A$ with atom-variables and introduces the nominal anti-unification problem with atom-variables following a semantic approach. Section 3 contains the rules for the AtomAntiUnif algorithm as well as arguments for its soundness, completeness and run-time complexity. Section 4 introduces a more expressive version of freshness constraints with atom-variables, and the AtomAntiUnifLgg algorithm that refines generalizations output by AtomAntiUnif, and computes a unique lgg of two $\mathtt{NL}_A$ terms-in-context. Proofs are available in the appendix.

# 2 Preliminaries

The nominal language $\mathtt{NL}_{aX}$ of expressions[1] is built by the grammar

$$S ::= a \mid f(S_1, \ldots, S_n) \mid \lambda a.S \mid \pi{\cdot}X \qquad\qquad \pi ::= \emptyset \mid (a\ b){\cdot}\pi$$

where $a, b$ are atoms in the infinite set *Atoms*, $\pi$ is a nonterminal for permutations, $(a\ b)$ is a swapping, $X$ is a nonterminal for generalization variables, $f, g$ are function symbols and $c, c_i$ are constant symbols in the function signature $\mathcal{F}$, where we assume that there is at least one (say $c$) of arity zero and one of arity 2. Compound expressions are function applications, and lambda-expressions which bind atoms. We also permit the tuple notation in examples. Applications in the lambda calculus can be represented using a binary function symbol.

The *ground language* $\mathtt{NL}_a$ is a sublanguage of $\mathtt{NL}_{aX}$ where variables $X$ and permutations $\pi$ are omitted. We consider only $\alpha$-equivalence $\sim$, which in fact is only defined on $\mathtt{NL}_a$.

An $\mathtt{NL}_{aX}$-*freshness constraint* is an expression of the form $a\#S$, expressing that $a$ is not free in (fresh for) $S$. We permit also $\bot$ as freshness constraint, which represents `False`. An $\mathtt{NL}_{aX}$-*freshness context* $(\nabla, \Delta, \ldots)$ is a set of $\mathtt{NL}_{aX}$-freshness constraints. We assume that permutation applications are homomorphically shifted inside expressions, where $\pi{\cdot}a$ is immediately computed. Every $\mathtt{NL}_{aX}$-freshness context can be transformed into a simpler one consisting only of constraints of the form $a\#X$ or $\bot$ by exhaustively using the rules:

$$\frac{a\#f(S_1,\ldots,S_n)}{a\#S_1,\ldots,a\#S_n} \quad \frac{a\#b}{} \quad \frac{a\#a}{\bot} \quad \frac{a\#\lambda b.S}{a\#S} \quad \frac{a\#\lambda a.S}{} \quad \frac{a\#\pi{\cdot}X}{\pi{\cdot}a\#X}$$

An $\mathtt{NL}_{aX}$-freshness context $\nabla$ is in *flattened form*, denoted by $\langle\nabla\rangle_{ff}$, when $\nabla$ is decomposed using the rules above, and permutations are eliminated such that only $a\#X$ and $\bot$ remain. A $\mathtt{NL}_{aX}$-freshness context is *consistent* if its flattened form does not contain $\bot$.

▶ **Definition 2.1** (Explanation of $\models$). *Let $\nabla$ be a consistent $\mathtt{NL}_{aX}$-freshness context, and $a\#S$ be a constraint. Then* $\nabla \models a\#S$ holds *iff* $\langle\{a\#S\}\rangle_{ff} \subseteq \langle\nabla\rangle_{ff}$.

An $\mathtt{NL}_{aX}$-*substitution* $\rho$ is a finite mapping from generalization variables to $\mathtt{NL}_{aX}$-expressions, extended to expressions. We will denote the domain of substitutions by $dom(.)$. A substitution is *ground* if it maps variables to $\mathtt{NL}_a$-expressions. For a ground substitution $\rho$: $\nabla\rho$ is called *valid* iff $\langle\nabla\rho\rangle_{ff}$ is consistent.

▶ **Lemma 2.2.** *Let $\nabla$ be an $\mathtt{NL}_{aX}$-freshness context, and $a\#S$ be an $\mathtt{NL}_{aX}$-freshness constraint. Then, $\nabla \models a\#S$ iff for all ground substitutions $\rho$: if $\nabla\rho$ is valid, then also $(\{a\#S\})\rho$ is valid.*

## 2.1 Nominal language $\mathtt{NL}_A$ with Atom-Variables

Let *AtomVars* be a set of atom-variables ranging over $A, B, A_1, B_1, \ldots$. The $\mathtt{NL}_A$-expression language is related to the nominal languages $\mathtt{NL}_a$ and $\mathtt{NL}_{aX}$, but includes *atom-variables* instead of atoms (see [24]). The grammar for the expression language $\mathtt{NL}_A$ of expressions $s$ and permutations $\pi$ with atom-variables is as follows:

$$s ::= W \mid \pi{\cdot}X \mid f(s_1, \ldots, s_n) \mid \lambda W.s \qquad W ::= \pi{\cdot}A \qquad \pi ::= \emptyset \mid (W_1\ W_2){\circ}\pi$$

Note that $\mathtt{NL}_A$-expressions do not contain atoms. There are two kinds of suspensions: of atom-variables, as in $\pi{\cdot}A$, and of generalization variables, as in $\pi{\cdot}X$. Composition of (atom-variable) permutations $\pi_1$ and $\pi_2$ is denoted $\pi_1 \circ \pi_2$ and can be flattened by concatenating

[1] The nominal language is equivalent to other nominal languages with different binding constructs.

the atom-variable swappings in $\pi_1, \pi_2$. For a permutation $\pi = \pi_1 \circ \ldots \circ \pi_n$, we write $\pi^{-1}$ for the inverse of $\pi$, i.e., for the permutation $\pi_n^{-1} \circ \ldots \circ \pi_1^{-1}$. Note that $\pi^{-1} = \pi$ holds for swappings $\pi$.

Substitutions are extended by a mapping of atom-variables.

The following notation will be used: $\mathit{Head}(s)$ is defined as $f$, if $s = f(\ldots)$; and $\lambda$, if $s = \lambda a.s'$; if $s$ is a suspension $\pi{\cdot}X$, then $X$; and if $s$ is $\pi{\cdot}A$ then $A$. We use the usual conventions for dealing with permutations and suspensions, for example, to move permutations homomorphically inside terms and viewing $\emptyset{\cdot}s$ as the same term as $s$. Suspensions of atom variables have to be treated carefully, for instance, $(A\ B) \cdot A = B$, for all instantiations of $A$ and $B$ to atoms, but $(A\ C) \cdot B$ is not necessarily $B$, since $B$, $C$ could be mapped to the same atom, resulting in $A$. We also will use the abbreviations $AV$ for atom-variables and $GV$ for generalization-variables.

▶ **Example 2.3.** Consider the $\mathtt{NL}_A$-expression $s = \lambda(A\ B){\cdot}C.f(C, X)$: it is an abstraction of the suspended atom-variable $(A\ B){\cdot}C$ on the expression $f(C, X)$. Let $\sigma = \{C \mapsto A, X \mapsto B\}$ be an $\mathtt{NL}_A$ substitution. Then, $s\sigma = \lambda(A\ B){\cdot}A.f(A, B) = \lambda B.f(A, B)$. For the ground substitution (i.e., mapping to $\mathtt{NL}_a$), $\rho = \{A \mapsto a_1, B \mapsto a_2, C \mapsto a_3, X \mapsto a_3\}$, we have: $s\rho = \lambda a_3.f(a_3, a_3)$.

Notice that the substitution $\sigma$ is applied to $s$ in a “capturing” way, as usual in nominal terms and nominal unification.

▶ **Definition 2.4.** $\mathtt{NL}_A$-freshness constraints *are pairs of the form $A\#s$ where $A$ is an atom-variable and $s$ is an $\mathtt{NL}_A$-expression.* $\mathtt{NL}_A$-freshness contexts *$(\nabla, \Delta, \ldots)$ are finite sets (conjunctions) of freshness constraints.*

In the following we permit $\pi{\cdot}A\#s$, but with the convention to replace $\pi{\cdot}A\#s$ immediately by $A\#\pi^{-1}{\cdot}s$, and also to move the permutation $\pi^{-1}$ inside the expression $s$. This move is done homomorphically, and stops at suspensions as follows: $\pi{\cdot}(\pi'{\cdot}V) \mapsto (\pi \circ \pi'){\cdot}V$.

▶ **Example 2.5.** It is possible to represent equality and inequality of atom-variables using $\mathtt{NL}_A$-freshness constraints: the constraint $A\#B$ means that instantiations of $A, B$ must be different. The constraint $A\#\lambda B.A$ enforces that the instantiations of $A, B$ must be equal. It is also possible to represent (a restricted form) of propositional formulas over equations on atom-variables, e.g., $(C = A) \vee (C = B)$ can be represented as $C\#\lambda A.\lambda B.C$.

$\mathtt{NL}_A$-freshness constraints and contexts can be further standardized/decomposed in the right hand side until they are of one of the following forms: $A\#\lambda W_1.\cdots.\lambda W_n.W$, or $A\#\lambda W_1.\cdots.\lambda W_n.\pi{\cdot}X$. Notice that $A\#A$ is inconsistent, but more complex constraints cannot be decomposed or evaluated without information about the concrete (i.e., ground) instances of the atom-variables. In the following we sometimes write freshness constraints/-contexts to mean $\mathtt{NL}_A$-freshness constraints/context.

▶ **Definition 2.6.** *A $\mathtt{NL}_A$-freshness constraint $A\#s$ is* valid *for a ground substitution $\rho$, iff $A\rho\#s\rho$ is valid in $\mathtt{NL}_a$. A freshness context $\nabla$ is* valid *for a ground substitution $\rho$, iff for every constraint $A\#s \in \nabla$, $(A\#s)\rho$ is valid. A freshness context $\nabla$ is* consistent*, iff there is a ground substitution $\rho$, such that $\nabla\rho$ is valid.*

▶ **Definition 2.7.** *Let $\nabla$ be an $\mathtt{NL}_A$-freshness context and $A\#s$ be a freshness constraint, and $\pi_1, \pi_2$ be permutations. Then*

- *$\nabla \models A\#s$ holds, iff for all ground substitutions $\rho$ and consistent $\nabla\rho$, $\nabla\rho \models (A\rho\#s\rho)$ holds.*
- *$\nabla \models \pi_1 = \pi_2$ holds, iff for all ground substitutions $\rho$ such that $\nabla\rho$ is consistent, $\nabla\rho \models \pi_1\rho = \pi_2\rho$ holds (as functions).*

We will define a more operational approach for deciding implication of $\mathtt{NL}_A$-freshness contexts and constraints, where the basic idea is to make a case analysis of all equal/inequal possibilities of atom-variables. We denote the set of atom-variables occurring in $\nabla$ as $\mathtt{AtVar}(\nabla)$ and write $\mathtt{AtVar}(\nabla, A\#s)$ for $\mathtt{AtVar}(\nabla \cup \{A\#s\})$. Also, $\mathtt{GenVar}(o)$ denotes the set of generalization variables of the object $o$.

▶ **Definition 2.8.** *Let $\nabla$ be a freshness context, and $R$ be an equivalence relation on $\mathtt{AtVar}(\nabla)$. An $R$-*realization function *$\rho_R$ is a function $\rho_R : \mathtt{AtVar}(\nabla) \to Atoms$, mapping every atom-variable $A$ in $\nabla$ to a concrete atom, such that $A_1 \sim_R A_2$ iff $\rho_R(A_1) = \rho_R(A_2)$.*

We assume that $\rho_R$ can be applied to substitutions by homomorphically applying it to the components. In an application $\rho_R(s)$, permutations are applied such that the result is an atom or a suspension $\pi{\cdot}X$ of a (generalization) variable $X$ where $\pi$ only contains atoms.

▶ **Definition 2.9.** *Let $\nabla$ be a freshness context and $A\#s$ be a freshness constraint. Then,* $\nabla \vdash_{ER} A\#s$ holds *iff for all equivalence relations $R$ on $\mathtt{AtVar}(\nabla, A\#s)$: $\rho_R(\nabla) \models \rho_R(A\#s)$.*

▶ **Lemma 2.10.** *Let $\nabla$ be a freshness context, and $A\#s$ be a freshness constraint. Then $\nabla \models A\#s$ iff $\nabla \vdash_{ER} A\#s$.*

▶ **Proposition 2.11.** *Let $\nabla, \Delta$ be freshness contexts. Then the complexity of the problem $\nabla \models \Delta$ is in coNP.*

**Proof.** We analyze the algorithm $\vdash_{ER}$ in Definition 2.9. We have to check for all equivalence-relations of atom-variables whether the relation holds. Since these equivalence-relations can be represented in polynomial size, and the test $\vdash$ is polynomial once the equivalence relation is chosen, we see that the problem is in coNP (see for example [15]). ◀

Alpha-equivalence on $\mathtt{NL}_A$ can be established semantically only: deciding whether all instances of, for example, $\lambda A.A$ and $\lambda B.(B\ C){\cdot}C$ are alpha-equivalent, intuitively takes us (using the usual nominal techniques) to checking whether $A \sim (B\ C){\cdot}C$ and $A\#C$, but the answer relies on all possible instantiations of $A, B$ and $C$.

▶ **Definition 2.12.** *An $\mathtt{NL}_A$-term-in-context is a pair $(\nabla, s)$ of an $\mathtt{NL}_A$-freshness context $\nabla$ and an $\mathtt{NL}_A$-expression $s$. The semantics of $(\nabla, s)$ is the set of (equivalence classes of) ground instances of $s$ that satisfy $\nabla$, i.e., $[\![(\nabla, s)]\!] := \{[r]_\sim \mid r$ is ground and $\exists\sigma : s\sigma \sim r \wedge \nabla\sigma$ holds$\}$, where $[r]_\sim$ denotes the equivalence class of $r$ modulo $\sim$.*

*A term-in-context $(\nabla, t)$ is* more general *than another term-in-context $(\nabla', t')$, if $[\![(\nabla', t')]\!] \subseteq [\![(\nabla, t)]\!]$. Two terms-in-context $(\nabla_1, t_1)$ and $(\nabla_2, t_2)$ are* equivalent *iff $[\![(\nabla_1, t_1)]\!] = [\![(\nabla_2, t_2)]\!]$. We will also write the equivalence of $(\nabla, t_1)$ and $(\nabla, t_2)$ as $\nabla \models t_1 = t_2$.*

We recall an example from [4] which are two $\mathtt{NL}_{aX}$ terms-in-context $(\{a\#X\}, f(X))$ and $(\{a\#X\}, f(a))$, where it is shown that $(\{a\#X\}, f(X))$ is not more general than $(\{a\#X\}, f(a))$. This behaviour is improved in our approach: after transferring their example into $\mathtt{NL}_A$, we have that $(\{A\#X\}, f(X))$ is more general than $(\{A\#X\}, f(A))$. In $(\{A\#X\}, f(X))$, the ground instance of $A$ can be chosen arbitrarily, thus $(\{A\#X\}, f(X))$ is equivalent to $(\emptyset, f(X))$.

▶ **Definition 2.13.** *A term-in-context $(\nabla, r)$ is called a* generalization *of two terms-in-context $(\nabla_1, s)$ and $(\nabla_2, t)$, if $[\![(\nabla_1, s)]\!] \subseteq [\![(\nabla, r)]\!]$ and $[\![(\nabla_2, t)]\!] \subseteq [\![(\nabla, r)]\!]$. It is a* least general generalization *(lgg) of $(\nabla_1, s)$ and $(\nabla_2, t)$ if it is a smallest one, i.e., for all least general generalizations $(\nabla', r')$ of $(\nabla_1, s)$ and $(\nabla_2, t)$, it holds $[\![(\nabla, r)]\!] \subseteq [\![(\nabla', r')]\!]$. A set $G$ of generalizations is* complete *iff for all generalizations $(\nabla', r')$ of $(\nabla_1, s)$ and $(\nabla_2, t)$, there is*

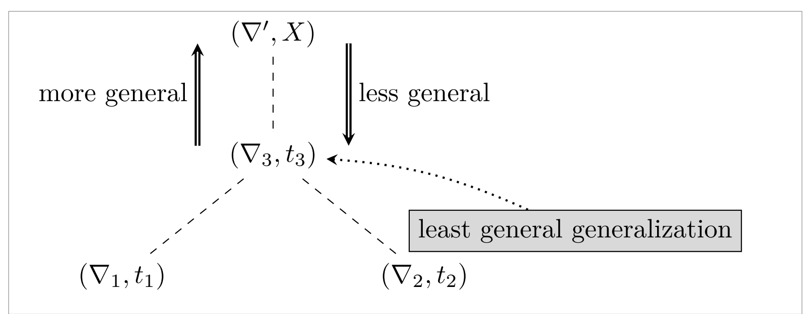
*some $g \in G$, such that $[\![g]\!] \subseteq [\![(\nabla', r')]\!]$. The set $G$ is* minimal*, if it is non-redundant, i.e. for all different $g_1, g_2 \in G$, $[\![g_1]\!] \not\subseteq [\![g_2]\!]$.*

*We call the generalization problem (of a language)* unitary *if there always exists a single lgg. We call it* finitary *or* infinitary *resp., if there always exists a minimal (finite, or unrestricted, resp.) complete set of generalizations for any input problem. If there are input problems such that a minimal complete set does not exist, the problem is called* nullary*. The latter case means that for the particular input problem, all complete sets are redundant.*

▶ **Example 2.14.** We reconsider the example of [4] of an infinite chain of generalizations, and show that the corresponding example for atom-variables becomes finite due to our semantics. Consider the sequence of terms-in-context $(\{A\#X\}, f(X, A))$; $(\{A\#X, B_1\#X, B_1\#A\}, f(X, A))$; $(\{A\#X, B_1\#X, B_1\#A,\ B_2\#X, B_2\#A,\ B_2\#B_1\},\ f(X, A))$ etc. In contrast to the framework in [4], the translated chain is not a counterexample to our claim, since the semantics remains constant within the chain: We will argue that $[\![\{A\#X\}, f(X, A)]\!]$ coincides with the set $[\![\{A\#X, B_1\#X, B_1\#A\}, f(X, A)]\!]$. Obviously $[\![\{A\#X, B_1\#X, B_1\#A\}, f(X, A)]\!] \subseteq [\![\{A\#X\}, f(X, A)]\!]$. Hence it is sufficient to show the converse inclusion. Let $\rho$ be such that $A\rho\#X\rho$. We simply have to show that the constraint $\{A\#X, B_1\#X, B_1\#A\}$ can be satisfied by defining $\rho$ appropriately on $B_1$. Since there are infinitely many atoms, there is always an atom not in $X\rho$, and also different from $A\rho$, say $b$. We define $B_1\rho := b$, and then the constraint is satisfied, and $f(X, A)\rho$ is an element of $[\![\{A\#X, B_1\#X, B_1\#A\}, f(X, A)]\!]$. Thus, the semantics of both terms-in-contexts is the same. The method can be used for all other terms-in-context of the translated chain and show that the chain is in fact constant, since there are infinitely many atoms available.

## 2.2 Nominal Anti-Unification Problem with Atom-Variables

The goal is to find a least general generalization of sets of $\mathtt{NL}_A$ terms-in-context.
**Problem**: Given two $\mathtt{NL}_A$ terms-in-context $(\nabla_1, t_1)$ and $(\nabla_2, t_2)$.
**Find:** A $\mathtt{NL}_A$-term-in-context $(\nabla_3, t_3)$ such that $[\![(\nabla_1, t_1)]\!] \subseteq [\![(\nabla_3, t_3)]\!]$ and $[\![(\nabla_2, t_2)]\!] \subseteq [\![(\nabla_3, t_3)]\!]$, and $[\![(\nabla_3, t_3)]\!]$ is as small as possible, i.e., $(\nabla_3, t_3)$ is a lgg of $(\nabla_1, t_1)$ and $(\nabla_2, t_2)$.




As example, let $(\{A_1\#A_2\}, f(A_1, \lambda A_2.A_2))$ and $(\emptyset, f(c, \lambda A_3.A_3))$ be $\mathtt{NL}_A$ terms-in-context, where $c$ is a unary symbol in the signature. Their least generalization is $(\emptyset, f(X, \lambda A_2.A_2))$, since the constraint does not restrict the instances. Another example for terms-in-context is $(\emptyset, f(A))$ and $(\emptyset, g(B))$. It is easy to check that $(\emptyset, X)$ is a generalization.

# 3 The Algorithm AtomAntiUnif

We first define the (non-deterministic) nominal generalization algorithm AtomAntiUnif that computes a generalization of the input terms-in-context. It relies on the subalgorithm Eqvm for equivariant matching (with atom-variables) that computes a permutation. We

prove that the algorithm is sound and complete, computes a generalization, and which can be performed in exponential time. Later in Section 4, we will define a method to compute a least general generalization from the computed generalization by adding a freshness context.

## 3.1 The Nominal Generalization Algorithm

The *state* of the algorithm AtomAntiUnif is a tuple $(\Gamma, M, \nabla, \theta)$ where

- $\Gamma$ is a set of generalization triples of the form $X : s \triangleq t$, where $X$ is a fresh (generalization-) variable, and $s, t$ are $\mathtt{NL}_A$-expressions.
- $M$ is a set of solved generalization triples.
- $\nabla$ is a set of freshness constraints.
- $\theta$ is a substitution represented as a list of bindings; the empty list is denoted as [].

The rules of the AtomAntiUnif, given in Figure 1, operate on such states. Given two $\mathtt{NL}_A$ expressions $s$ and $t$, and a freshness context $\nabla$ (possibly empty), to compute generalizations for $(\nabla, s)$ and $(\nabla, t)$, we start with $(\{X : s \triangleq t\}, \emptyset, \nabla, [])$, the *initial state* (sometimes we abbreviate it to $(\nabla, \{X : s \triangleq t\})$), where $X$ is a fresh generalization variable, and we apply the rules as long as possible, until no more rule applications are possible, where no alternative rule applications have to be explored. The *final state* will be reached, which has the form $(\emptyset, M, \Delta, \theta)$. We will denote the computation from the initial state to the final state as $(\Gamma, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$. When convenient we will denote by $(\Gamma, M, \nabla, \theta) \Longrightarrow_{(\mathtt{R})} (\Gamma', M', \nabla', \theta')$, the one-step computation using a rule (R) from Figure 1. The output is a term-in-context obtained from the generated substitution $\theta$ and the final freshness constraint $\Delta$, i.e. the output is $(\Delta, X \circ \theta)$, also called the *result computed* by the AtomAntiUnif algorithm.

We now describe the rules in Figure 1:

- The decomposition rule (`Dec`) is standard.
- Rule (`Abs`) is applicable for the generalization (on variable $X$) of two generalized abstractions: apply a swapping with a fresh atom-variable $B$. Its freshness is guaranteed by adding freshness constraints to $\nabla'$, which guarantee that the renaming by the permutation keeps $\alpha$-equivalence, without losing solutions. The substitution is extended with a mapping $\{X \mapsto \lambda B.Y\}$, where $Y$ is a fresh generalization variable.
- Rules (`SusA`) and (`SusYY`) are applicable for the generalization (on variable $X$) of suspensions of atom-variables or generalization variables, respectively. In both cases, the semantics of $\nabla$ will identify the two suspensions, and the substitution will be extended with a mapping from $X$ to one of the variables in the suspension. Note that only these two rules introduce the variables from the initial $s, t$ into the solution substitution $\theta$.
- Similarly to [4], the merging rule (`Mer`) relies on a generalization (to $\mathtt{NL}_A$) of the equivariance algorithm Eqvm, for computing a generalized permutation $\pi$, such that $\nabla \models \pi{\cdot}(s_1, t_1) = (s_2, t_2)$, if it exists.
- Rules (`Solve`), (`SolveYY`) and (`SolveAB`) can be applied when a condition is satisfied. The first rule applies when $s$ and $t$ do not have the same head, and are not suspensions of atom variables. Rule (`SolveYY`) treats the case of two suspensions on the same generalization variable $Y$, say $\pi_1{\cdot}Y$ and $\pi_2{\cdot}Y$, but the semantics of $\nabla$ permits that both suspensions may be mapped to different atoms. Rule (`SolveAB`) treats the case of suspensions of atom-variables, in the case where the semantics of $\nabla$ implies that both suspensions will be instantiated to two different terms. The algorithm then ignores the constraints acting on the variables in the equations that are moved to $M$.

(Dec): Decomposition

$$\frac{\{X{:}f(s_1,\dots,s_n) \triangleq f(t_1,\dots,t_n)\}\dot\cup\Gamma, M, \nabla, \theta}{\Gamma\dot\cup\{X_1{:}s_1 \triangleq t_1,\dots,X_n{:}s_n \triangleq t_n\}, M, \nabla, \theta\cup\{X\mapsto f(X_1,\dots,X_n)\}}$$ where $X_i$ are fresh variables

(Abs): Abstraction

$$\frac{\{X{:}\lambda W_1.s \triangleq \lambda W_2.t\}\dot\cup\Gamma, M, \nabla, \theta}{\Gamma\dot\cup\{Y{:}(W_1\ B)\cdot s \triangleq (W_2\ B)\cdot t\}, M, \nabla\cup\{B\#\lambda W_1.s, B\#\lambda W_2.t\}, \theta\cup\{X\mapsto \lambda B.Y\}}$$

where $Y$ is a fresh variable, and $B$ is a fresh atom-variable

(SusA): SuspensionA

$$\frac{\{X{:}W_1 \triangleq W_2\}\dot\cup\Gamma, M, \nabla, \theta \quad \nabla\vDash W_1 = W_2}{\Gamma, M, \nabla, \theta\cup\{X\mapsto W_1\}}$$

(SusYY): SuspensionYY

$$\frac{\{X{:}\pi_1\cdot Y \triangleq \pi_2\cdot Y\}\dot\cup\Gamma, M, \nabla, \theta \quad \nabla\vDash \pi_1 = \pi_2}{\Gamma, M, \nabla, \theta\cup\{X\mapsto \pi_1\cdot Y\}}$$

(Mer): Merging

$$\frac{\Gamma, \{Z_1{:}s_1 \triangleq t_1, Z_2{:}s_2 \triangleq t_2\}\dot\cup M, \nabla, \theta \quad \textsc{EqvM}(\{(s_1,t_1)\preceq(s_2,t_2)\},\nabla) = \pi}{\Gamma, M\cup\{Z_1{:}s_1 \triangleq t_1\}, \nabla, \theta\cup\{Z_2\mapsto \pi\cdot Z_1\}}$$ where $(Z_1,Z_2)$ is $(X,Y)$ or $(A,B)$

(Solve)

$$\frac{\{X{:}s \triangleq t\}\dot\cup\Gamma, M, \nabla, \theta}{\Gamma, M\cup\{X{:}s \triangleq t\}, \nabla, \theta}$$ If $Head(s) \neq Head(t)$ and if $s$ and $t$ are not both suspensions of atom-variables.

(SolveYY)

$$\frac{\{X{:}\pi_1\cdot Y \triangleq \pi_2\cdot Y\}\dot\cup\Gamma, M, \nabla, \theta \quad \nabla\not\vDash \pi_1 = \pi_2}{\Gamma, M\cup\{X{:}\pi_1\cdot Y \triangleq \pi_2\cdot Y\}, \nabla, \theta}$$

(SolveAB)

$$\frac{\{X{:}W_1 \triangleq W_2\}\dot\cup\Gamma, M, \nabla, \theta \quad \nabla\not\vDash W_1 = W_2}{\Gamma, M\cup\{A{:}W_1 \triangleq W_2\}, \nabla, \theta\cup\{X\mapsto A\}}$$

$A$ is a fresh atom-variable.

**Figure 1** Rules of the algorithm AtomAntiUnif.

- The generalization variables are always chosen fresh, thus these do not occur in freshness contexts, hence application of $\theta$ to $\nabla$ is not necessary. Note that the effect of the rules on $\nabla$ is only the addition of constraints by (Abs) (without a semantical change).
- By construction, in a computation $(\{X : s \triangleq t\}, \emptyset, \nabla, []) \Longrightarrow^* (\Gamma, M, \nabla, \theta)$ from the initial state to an intermediate state, the generalization variables in the range of the computed substitution $\theta$ satisfy $\texttt{GenVar}(\theta) \subseteq \texttt{GenVar}(\Gamma\cup M)$.

▶ Remark 3.1 (About completeness). Several rules do not inherit the optimal set of freshness constraints. It is unclear how and presumably complex to compute these. We leave this computation to the extra algorithm AtomAntiUnifLgg that computes a lgg from the result of AtomAntiUnif by adding and checking generalized freshness constraints (see Section 4)

▶ Remark 3.2. An algorithm for $\nabla\vDash\dots$ is to check all equivalence classes of atom-variables, where equivalence of $A, B$ semantically means equal images under a ground instantiation (see also Definition 2.8, Lemma 2.10, and Section 4). This can be performed in exponential time since it is sufficient to check all possibilities of equality and disequality of atom-variables.

The next example shows the use of the semantics of $\nabla$ and the treatment of bindings.

▶ **Example 3.3.** Let the $\texttt{NL}_A$-expressions to be generalized be $\lambda A_1.A_2$ and $\lambda A_2.A_1$ under $\nabla$. The lgg is $(\emptyset, \lambda A.A)$ or $(\emptyset, \lambda A.B)$ for $A\neq B$, depending on whether $\nabla\vDash A = B$ or $\nabla\not\vDash A = B$. In the first case (Abs) is applied and in the latter case also (SusA).

The algorithm AtomAntiUnif is naive: it finds a generalization for two $\texttt{NL}_A$-terms-in-context $(\nabla, s)$ and $(\nabla, t)$ but not necessarily the least general one:

$$\frac{\Psi \dot\cup \{e \preceq e\}, \Pi, \nabla}{\Psi, \Pi, \nabla} \qquad \frac{\Psi \dot\cup \{W_1 \preceq W_2\}, \Pi, \nabla}{\Psi, \{W_1 \mapsto W_2\} \cup \Pi, \nabla} \qquad \frac{\Psi \dot\cup \{(f\ s_1 \ldots s_n) \preceq (f\ s'_1 \ldots s'_n)\}, \Pi, \nabla}{\Psi \cup \{s_1 \preceq s'_1, \ldots, s_n \preceq s'_n\}, \Pi, \nabla}$$

$$\frac{\Psi \dot\cup \{\pi_1{\cdot}X \preceq \pi_2{\cdot}X\}, \Pi, \nabla \quad \nabla \models \pi_1{\cdot}X = \pi_2{\cdot}X}{\Psi, \Pi, \nabla} \qquad \frac{\Psi \dot\cup \{\lambda W_1.s \preceq \lambda W_2.t\}, \Pi, \nabla \quad \nabla \models W_2 \# \lambda W_1.s}{\Psi \cup \{(W_1\ W_2){\cdot}s \preceq t\}, \Pi, \nabla}$$

$$\frac{\Psi \dot\cup \{\lambda W_1.s \preceq \lambda W_2.t\}, \Pi, \nabla \quad \nabla \models W_1 \# \lambda W_2.t}{\Psi \cup \{s \preceq (W_1\ W_2){\cdot}t\}, \Pi, \nabla} \qquad \frac{\emptyset, \Pi, \nabla \qquad \textsc{EqvBiEx}(\Pi, \nabla) = \pi}{\text{Return } \pi}$$

**Figure 2** Rules of the permutation matching algorithm EQVM.

$$\frac{\nabla \models W_1 = W'_1 \wedge W_2 = W'_2 \quad (\Pi \dot\cup \{W_1 \mapsto W_2, W'_1 \mapsto W'_2\}, \nabla)}{(\Pi \dot\cup \{W_1 \mapsto W_2\}, \nabla)} \qquad \frac{\nabla \models W_1 = W'_1, \quad \nabla \not\models W_2 = W'_2 \quad (\Pi \dot\cup \{W_1 \mapsto W_2, W'_1 \mapsto W'_2\}, \nabla)}{\bot}$$

$$\frac{\nabla \models W_2 = W'_2, \nabla \not\models W_1 = W'_1 \quad (\Pi \dot\cup \{W_1 \mapsto W_2, W'_1 \mapsto W'_2\}, \nabla)}{\bot} \qquad \frac{\Pi, \nabla \qquad \text{no other rule is applicable}}{\text{Return a permutation computed from } \Pi}$$

**Figure 3** Rules of the bijection extraction algorithm EQVBIEX.

▶ **Example 3.4.** Consider the $\mathtt{NL}_A$-terms-in-context $(\emptyset, f(c_1, A))$ and $(\emptyset, f(c_2, A))$ that have to be generalized, where $c_1$ and $c_2$ are different constant symbols. The generalization computed by the ATOMANTIUNIF algorithm is $(\emptyset, f(X', A))$, obtained via the derivation: $(\{X : f(c_1, A) \triangleq f(c_2, A)\}, \emptyset, \emptyset, []) \Longrightarrow^* (\emptyset, \{X' : c_1 \triangleq c_2\}, \emptyset, \{X \mapsto f(X', A)\})$ However, this is not the smallest generalization, since $(\{A\#X'\}, f(X', A))$ has a smaller set of instances, and is less general. The reason that this construction works is that $A$ is a part of the solution, but it does not occur in the solved equation associated to $X'$.

Notice, however, that $f(A, A)$ and $f(c, A)$ have a generalization $(\emptyset, f(X', A))$, via the derivation: $(\{X : f(A, A) \triangleq f(c_2, A)\}, \emptyset, \emptyset, []) \Longrightarrow^* (\emptyset, \{X' : A \triangleq c_2\}, \emptyset, \{X \mapsto f(X', A)\})$. But $\{A\#X'\}$ cannot be added as freshness context: although $A$ is part of the solution, $A$ occurs in the solved equation associated to $X'$. In fact, $(\{A\#X'\}, f(X', A))$ is not a generalization of $f(A, A)$ and $f(c, A)$, and $(\emptyset, f(X', A))$ is the lgg.

## 3.2 Equivariance Algorithm

The merge-rule as in [4] relies on solving an equivariance problem [9]. It will be treated similarly here, however, generalized to atom-variables and nested permutations. We will use a matching-like rule-based algorithm that finally is able to produce a permutation for the merge rule, if there is one at all, and otherwise fails. Instead of fixing a derivation algorithm $\nabla \vdash \ldots$, we will use the semantic variant $\nabla \models \ldots$ and mean it to be as general as possible, which will leave some open space for optimizations.

▶ **Algorithm 3.5.** *The rules of the non-deterministic algorithm* EQVM *are in Figures 2 and 3. They operate on triples of the form* $(\Psi, \Pi, \Delta)$ *where*

- $\Psi$ *is a set of matching problems of the form* $e \preceq e'$*;*
- $\Pi$ *are the potential (mapping) components of the permutation computed so far;*
- $\Delta$ *is set of freshness constraints.*

*The initial triple is* $(\Psi, \emptyset, \nabla)$*, where* $\Psi$ *and* $\nabla$ *are delivered in the call to this algorithm. The rules are to be applied as long as possible. If a state of the form* $(\emptyset, \Pi, \Delta)$ *is reached, and* $\Delta$ *implies that the mappings in* $\Pi$ *do not collide and can be completed into a permutation*

*(a bijection), then the algorithm is successful. The permutation will be computed using the algorithm* EqvBiEx *in Figure 3. In the success case a permutation $\pi$ from $\Pi$ is returned after an exhaustive run without a fail, where for the permutation perhaps some mappings have to be added. For example, the result $\{A \mapsto C, B \mapsto D, C \mapsto E\}$ is made a bijection by adding $\{D \mapsto A, E \mapsto B\}$, which can then be represented as a permutation.*

▶ **Proposition 3.6.** *The algorithm* EqvM *is sound, correct and terminates in a linear number of steps with a computed permutation if there is any.*

**Proof.** Every step is easily justified and makes the set $\Psi$ strictly smaller. The major parts of the complexity are the calls to $\nabla \vDash$. For the complexity of $\nabla \vDash \dots$ see Proposition 2.11. ◀

## 3.3 Properties of AtomAntiUnif

From an AtomAntiUnif derivation $(\{X : s \triangleq t\}, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$ we can obtain substitutions $\rho_1$ and $\rho_2$ mapping generalization variables $\texttt{GenVar}(\theta)$ to $\texttt{NL}_A$-expressions, making a connection between $X \circ \theta$, $s$ and $t$ (similar as in [4]). From the set of solved equations $M = \{X_1 : s_1 \triangleq t_1, \dots, X_n : s_n \triangleq t_n\}$, we define $\rho_1 = \{X_1 \mapsto s_1, \dots, X_n \mapsto s_n\}$ and $\rho_2 = \{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$, such that $[\![(\nabla, s)]\!] \subseteq [\![(\Delta, (X \circ \theta)\rho_1)]\!]$ and $[\![(\nabla, t)]\!] \subseteq [\![(\Delta, (X \circ \theta)\rho_2)]\!]$. In general, $[\![(\nabla, s)]\!] \cup [\![(\nabla, t)]\!] \subseteq [\![(\Delta, (X \circ \theta)\rho_1)]\!] \cup [\![(\Delta, (X \circ \theta)\rho_2)]\!]$ where this is also specialized to the intermediate states of AtomAntiUnif.

This connection motivates an extension of the semantics for $\texttt{NL}_A$-terms-in-context $(\nabla, s)$ to AtomAntiUnif states, which will allow us to show that the semantics increase in each step of rule application.

▶ **Definition 3.7.** *Let $(\{X : s \triangleq t\}, \emptyset, \nabla, [])$ be an initial state of* AtomAntiUnif*. The semantics of terms-in-context w.r.t. states $S := (\Gamma, M, \nabla, \theta)$ of* AtomAntiUnif *as follows:*

- $[\![(\{X : s \triangleq t\}, \emptyset, \nabla, [])]\!] := [\![(\nabla, s)]\!] \cup [\![(\nabla, t)]\!]$
- $[\![(\emptyset, \emptyset, \nabla, \theta)]\!] := [\![(\nabla, X \circ \theta)]\!]$*, where $X$ is the input generalization variable.*
- *Otherwise, for $\Gamma \cup M = \{Y_i : s_i \triangleq t_i \mid i = 1, \dots, n\} \neq \emptyset$:*

$$[\![(\Gamma, M, \nabla, \theta)]\!] := \{[(X \circ \theta)\rho_s]_\sim \mid \rho_s \text{ is ground}, \nabla\rho_s \text{ holds, and } \forall i.[Y_i\rho_s]_\sim \in [\![(\nabla, s_i)]\!]\} \cup \{[(X \circ \theta)\rho_t]_\sim \mid \rho_t \text{ is ground}, \nabla\rho_t \text{ holds, and } \forall i.[Y_i\rho_t]_\sim \in [\![(\nabla, t_i)]\!]\}$$

*where $X$ is the input generalization variable.*

▶ **Proposition 3.8.** *The algorithm* AtomAntiUnif *never gets stuck and will yield a generalization. The number of rule applications is linear.*

As illustrated in previous examples, our algorithm outputs a generalization but not always the least general one (cf. Example 3.4). Therefore, we can establish the following weaker completeness theorem that establishes a characterization of lggs up to a freshness context.

▶ **Theorem 3.9** (Completeness up to Freshness Contexts)**.** *Given $\texttt{NL}_A$ expressions $s$ and $t$, and a freshness context $\nabla$. If $(\nabla', r)$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$, then there exists a $\nabla''$ and a derivation $(\{X : s \triangleq t\}, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$ such that $[\![(\Delta \cup \nabla'', X \circ \theta)]\!]$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$, and $[\![(\emptyset, M, \Delta \cup \nabla'', \theta)]\!] \subseteq [\![(\nabla', r)]\!]$.*

**Proof.** The proof follows by induction on the number of rule applications from Figure 1. The complete proof can be found in the appendix. ◀

Soundness of AtomAntiUnif is obtained by an easy inspection of the rules in Figure 1 and the proof is omitted.

▶ **Theorem 3.10** (Complexity of the Algorithm). *The algorithm* AtomAntiUnif *requires simple exponential time to compute a solution. The number of rule applications is polynomial, and the solution requires polynomial space.*

**Proof.** The number of rule applications of the main algorithm is polynomial, since the size of $\Gamma \cup M$ is strictly reduced in each step, where the *size* is meant as follows: the term size ignoring the permutations, and a generalization variable has size 2, whereas an atom-variable has size 1. The total size, including permutations, remains polynomial, since there is only a constant-size increase per rule application in $\Gamma, M, \nabla$. The size of $\theta$ also remains polynomial, since solution components are not applied. The call to subalgorithms may be (simply) exponential depending on the size of the problem. Since there is only a polynomial number of such calls, and the size remains polynomial, the algorithm requires simple exponential time in the worst case. ◀

The following shows that the $\texttt{NL}_A$-freshness contexts have a finiteness property, where an upper bound is given. We consider two freshness constraints $\nabla_1, \nabla_2$ as equivalent w.r.t. $s$ iff $[\![(\nabla_1, s)]\!] = [\![(\nabla_2, s)]\!]$.

▶ **Theorem 3.11.** *Let $s$ be a generalization term. Then the number of equivalence classes of $\texttt{NL}_A$-freshness constraints $\nabla$ w.r.t. $s$ is finite. Their number is in $O(n^n) = O(e^{n*(log(n)+1)})$.*

**Proof.** Let $M = \texttt{AtVar}(s)$, and let $\nabla_1$ and $\nabla_2$ be freshness contexts, where we assume that freshness constraints are of the two forms $A\#\lambda W_1.\dots.\lambda W_n.W$ and $A\#\lambda W_1.\dots.\lambda W_n.\pi{\cdot}X$.
**(Equivalence test)** The test for equivalence of two freshness contexts $\nabla_1$ and $\nabla_2$ is as follows: For all equivalence relations $R$ on $M$ and for all freshness contexts $\Delta_0 \subseteq \{A\#X \mid A \in \texttt{AtVar}(s), X \in \texttt{GenVar}(s)\}$:

- Check the equivalence of $\nabla_1$ and $\nabla_2$ (under $R$ and $\Delta_0$) of the following two tests:
  - **(1)** There is an equivalence relation $R'$ on $\texttt{AtVar}(\nabla_1, s)$ that soundly extends $R$, i.e. $A \sim_R A'$ iff $A \sim_{R'} A'$ for all atom-variables $A, A'$ in $M$. Such that:
    - **(i)** All AV-constraints $A\#\lambda W_1.\dots.\lambda W_n.W$ in $\nabla_1$ evaluate to true under $R'$.
    - **(ii)** The set of GV-constraints $\nabla_G$ in $\nabla_1$ is equivalent to $\Delta_0$ using $R'$ for simplification.
  - **(2)** There is an equivalence relation $R''$ on $\texttt{AtVar}(\nabla_2, s)$ that soundly extends $R$, i.e. $A \sim_R A'$ iff $A \sim_{R''} A'$ for all atom-variables $A, A'$ in $\texttt{AtVar}(\nabla_2, s)$. Such that:
    - **(i)** All AV-constraints $A\#\lambda W_1.\dots.\lambda W_n.W$ in $\nabla_2$ evaluate to true under $R''$.
    - **(ii)** The set of GV-constraints $\nabla_G$ in $\nabla_2$ is equivalent to $\Delta_0$ using $R'$ for simplification.
- If for all $R$, and all selected freshness contexts $\Delta_0$, the combined tests leads to true, then the freshness contexts $\nabla_1$ and $\nabla_2$ are equivalent w.r.t. $s$.

(**Finiteness)** of the set of equivalence classes of freshness constraints follows, since the set $M$ depends only on $s$, and hence there is a finite set of equivalence relations over $M$. Also, there is only a finite set of possibilities for $\Delta_0$. An upper bound for the number of equivalence classes $R$ is $O(n^n) = O(e^{(n*log(n))})$, where $n$ is the number of atom-variables in $s$. (The number of equivalence relations for $n$ atom-variables is also called the Bell-number[2] of $n$.)

The number of different sets of freshness constraints is at most exponential, i.e. $O(2^n)$ where $n$ is the number of atom-variables, since these are all subsets of a set of atom-variables. Since both are combined, the number of possibilities is their product, and thus the growth is in $O(e^{(n*(log(n)+1))}$. ◀

[2] see `https://mathworld.wolfram.com/BellNumber.html`.

▶ **Corollary 3.12.** *Let $(\nabla, \Gamma)$ be an input for the* AtomAntiUnif *algorithm and $(\nabla', s)$ be the computed result. Then, there are no infinitely properly descending chains $(\nabla', s)$, $(\nabla'_1, s)$, $(\nabla'_2, s), \ldots$ of generalizations. A consequence is that the solution type of generalization problems with atom-variables is finitary or even unitary.*

# 4 Computing Least General Generalizations

In this section we describe the final step of computing lggs, which builds upon the result of the algorithm AtomAntiUnif. First, we define a more general form of freshness constraints. Then, we provide a specialized algorithm, called AtomAntiUnifLgg, that computes an lgg by strengthening the (general) freshness constraints.

Notice that the addition of constraints of the form $A\#X$ is not sufficient to reach an lgg:

▶ **Example 4.1.** Let the input be $(\emptyset, \{X : (f(A), A, B) \triangleq (c, A, B)\})$, where $c, f$ are function constants. AtomAntiUnif computes the generalization $(\emptyset, (Y, A, B))$, however, it is not an lgg. Adding $A\#Y$ or $B\#Y$ violates the generalization property. However, $B\#\lambda A.Y$ can be added as constraint, and then $(\{B\#\lambda A.Y\}, (Y, A, B))$ allows the instance $(f(a), a, a)$, since $a\#\lambda a.f(a)$ holds. We see that $(\{B\#\lambda A.Y\}, (Y, A, B))$ is the lgg.

## 4.1 A Generalized Representation for lgg

In order to represent and compute lggs, we switch to a more general representation of freshness constraints and contexts, called EqR-*freshness constraints*, that are obtained by analyzing the possible EQuivalence Relations over a set of atom-variables $\mathcal{A}$. We will show that a least general generalization using EqR-*freshness contexts* can be obtained and how to compute it. The extended form allows for a finite generation of extra freshness constraints and also a computable test of whether the extended solution still is a generalization. Note that this is the same as making a complete case-distinction over all possible equality/disequalities of ground instantiations of atom-variables.

▶ **Definition 4.2.** *Let $\mathcal{A} = \{A_1, \ldots A_n\}$ be a set of atom-variables and $\{Y_1, \ldots, Y_m\}$ be a set of (generalization) variables.*

- *An* EqR-freshness constraint *is of the form $(Q \implies Q')$ where $Q$ is a conjunction of constraints of the forms $A \neq A'$ or $A = A'$. The part $Q'$ is* `True`*, or* `False`*, or a (positive) propositional formula formed using $\wedge$,$\vee$, and freshness constraints of the form $A_i\#Y_j$.*
- *An* EqR-(freshness) context *is a conjunction of* EqR*-constraints, i.e., it has the form $(Q_1 \implies Q'_1) \wedge \ldots \wedge (Q_m \implies Q'_m)$ where the $Q_i$ are exactly all equivalence relations within $\mathcal{A}$. Notice that if $A = B$ holds in $Q_i$, then the representation is not unique and could be extended or restricted, e.g., by replacing $A\#X$ by $B\#X$ or by similar operations.*

▶ **Example 4.3.** Let $\mathcal{A} = \{A, B\}$, and $X$ be a variable. An EqR- freshness context is: $(\{A = B\} \implies (A\#X \wedge B\#X)) \wedge (\{A \neq B\} \implies (B\#X))$.
A second example is the constraint $A\#\lambda B.A$ that has an equivalent EqR-freshness context: $(\{A = B\} \implies \texttt{True}) \wedge (\{A \neq B\} \implies \texttt{False})$.
A third example is the freshness constraint $A\#\lambda B.(B\ A){\cdot}Y$ with $\mathcal{A} = \{A, B\}$. The equivalent EqR-freshness context is $(A = B \implies \texttt{True}) \wedge (A \neq B \implies B\#Y)$.

▶ **Definition 4.4.** *For an equivalence relation $R$ on $\mathcal{A}$, we define a related evaluation of freshness constraints, denoted as $\gamma(R, A\#s)$, which exploits the equalities and inequalities of a single $R$ to maximally simplify the constraint(s), and thereby eliminating all permutations by perhaps applying the inverse if necessary, such that the result is only* `True`, `False`*, or a freshness constraint of the form $A\#B$, or $A\#Y$.*

The next result establishes an action of $\gamma(R)$ over freshness constraints, connecting them to EQR-freshness constraints.

▶ **Proposition 4.5.** *Every freshness context can also be expressed as an* EQR*-freshness context. The size of the generated output may be exponential.*

There exist EQR-freshness contexts that cannot be encoded as $\mathtt{NL}_A$-freshness context:

▶ **Lemma 4.6.** *Let $\mathcal{A} = \{A, B\}$ be a set of atom-variables, $X$ and $Y$ be generalization variables. The* EQR*- freshness context $((A \neq B) \implies A\#X) \wedge ((A = B) \implies A\#Y \wedge B\#Y)$ cannot be encoded as an $\mathtt{NL}_A$-freshness context, i.e, a conjunction of $\mathtt{NL}_A$-freshness constraints.*

▶ **Corollary 4.7.** EQR*-contexts are strictly more expressive than $\mathtt{NL}_A$-freshness contexts.*

The next example illustrates (i) the evaluation of a freshness constraint under an equivalence relation $R$ in a term-in-context (ii) the omission of a redundant atom-variable, and is an example where a single least general generalization does not exist.

▶ **Example 4.8.** Let $\mathcal{A} = \{A, B, C\}$ and $\nabla = \{C\#\lambda A.\lambda B.C, C\#X\}$ be a freshness context. A first generalization of $(\nabla, X' : f(A, B, C) \triangleq f(A, B, X))$ is the term-in-context $(\nabla, f(A, B, X))$. An EQR-freshness context $\nabla_0$ representing this (obtained using Definition 4.2) is as follows:

| | | | | | |
|---|---|---|---|---|---|
| 1) | | $(\{A = B, B = C, A = C\}$ | $\implies$ | $(C\#X))\}$ | (or $A\#X$) |
| 2) | $\wedge$ | $(\{A = B, B \neq C, A \neq C\}$ | $\implies$ | $\mathtt{False})$ | |
| 3) | $\wedge$ | $(\{A \neq B, B \neq C, A \neq C\}$ | $\implies$ | $\mathtt{False})$ | |
| 4) | $\wedge$ | $(\{A \neq B, B = C, A \neq C\}$ | $\implies$ | $(C\#X))$ | (or $B\#X$) |
| 5) | $\wedge$ | $(\{A \neq B, A = C, B \neq C\}$ | $\implies$ | $(C\#X))$ | (or $A\#X$) |

Some computations (see also Proposition 4.9 for the general case) show that $(\{A = B\} \implies A\#X) \wedge (\{A \neq B\} \implies (B\#X \vee A\#X))$ is an equivalent EQR-freshness context for the set $\{A, B\}$, which cannot be represented as an $\mathtt{NL}_A$-freshness context using only $\{A, B\}$ due to the disjunction. This is the (unique) lgg w.r.t. EQR-freshness contexts.

Notice that a complete set consists of two lggs w.r.t. $\mathtt{NL}_A$-freshness contexts: $(\{A\#X\}, f(A, B, X))$ and $(\{B\#X\}, f(A, B, X))$, if only atom-variables from $f(A, B, X)$ are permitted. However, using additional atom-variables (here $C$), we obtain a unique lgg $(\nabla, f(A, B, X))$. This example provides an argument for permitting a larger set of atom-variables as the ones contained in the term of the term-in-context for obtaining lggs w.r.t. $\mathtt{NL}_A$-freshness contexts.

▶ **Proposition 4.9** (Restricting). *Let $\mathcal{A}' \subset \mathcal{A}$ be a set of atoms, $X_1, \ldots, X_n$ be variables, $t$ be an expression with $\mathtt{AtVar}(t) \subseteq \mathcal{A}'$, and $\nabla$ be a freshness context over $\mathcal{A}$. Then there is an equivalent* EQR*-freshness context $\nabla'$ over $\mathcal{A}'$, such that $[\![(\nabla, t)]\!] = [\![(\nabla', t)]\!]$.*

## 4.2 The AtomAntiUnifLgg algorithm

This algorithm strengthens the freshness constraints of the computed generalization given as output by the AtomAntiUnif algorithm. It takes as input $(\nabla_{in}, Y{:}r_1 \triangleq r_2)$ and $(\nabla_{\mathrm{AAU}}, t)$, where $(\nabla_{in}, Y{:}r_1 \triangleq r_2)$ is the input of AtomAntiUnif with result $(\nabla_{\mathrm{AAU}}, t)$, and $\nabla_{\min}$ is the result of restricting $\nabla_{\mathrm{AAU}}$ to the atom-variables in $t$ (cf. Proposition 4.9). Then all atom variables occurring in $\nabla_{\min}$ also occur in $(\nabla_{in}, Y : r_1 \triangleq r_2)$. The output will be the freshness context $\nabla_{\min}$ joined with a generalized freshness context $\mathcal{Q}$.

▶ **Definition 4.10.** *The algorithm* AtomAntiUnifLgg *for computing a least general generalization is defined in Figure 4.*

1: **Input:** $(Y : r_1 \triangleq r_2, \nabla_{in}), (\nabla_{\mathrm{AAU}}, t)$
2: Let $\mathcal{A} = \mathtt{AtVar}(t)$.
3: Let $\nabla_{\min}$ be the result of restricting $\nabla_{\mathrm{AAU}}$ to $\mathcal{A}$ as in (the algorithm from) Proposition 4.9.
4: $\mathcal{X} = \mathtt{GenVar}(t)$; $\mathcal{Q} := \mathtt{True}$
5: **for** every equivalence relation $R$ on $\mathcal{A}$ **do** ▷ Computation of $\mathcal{Q}$
6: let $Q_1 = \gamma(R)$
7: let $Q_2$ be the maximal possible positive formula of GV-constraints $A\#X$ with $A \in \mathcal{A}$
8: and $X \in \mathcal{X}$ such that the following holds:

**a)** $[\![(\nabla_{in}, r_1)]\!] \subseteq [\![((Q_1 \implies Q_2) \cup \nabla_{\min}), t)]\!]$.
**b)** $[\![(\nabla_{in}, r_2)]\!] \subseteq [\![((Q_1 \implies Q_2) \cup \nabla_{\min}), t)]\!]$.

9: Let $Q_2'$ and $\mathcal{Q}'$ be such that $\mathcal{Q} = (\mathcal{Q}' \wedge (Q_1 \implies Q_2'))$
10: $\mathcal{Q} := (\mathcal{Q}' \wedge (Q_1 \implies (Q_2' \wedge Q_2)))$
11: **end for**
12: **Output:** $(\nabla_{\min} \cup \mathcal{Q}, t)$

**Figure 4** The AtomAntiUnifLgg Algorithm.

▶ **Proposition 4.11** (Unique Maximal Constraint). *Let $(\nabla, t)$ be a term-in-context with* EQR*-freshness context $\nabla$, where $\mathtt{AtVar}(\nabla) \subseteq \mathtt{AtVar}(t)$ and $\mathtt{GenVar}(\nabla) \subseteq \mathtt{GenVar}(t)$, and let $(\nabla_1, r_1)$, $(\nabla_1, r_2)$ be terms-in-context with $\mathtt{AtVar}(r_1, r_2) \subseteq \mathtt{AtVar}(t)$, $\mathtt{GenVar}(r_1, r_2) \subseteq \mathtt{GenVar}(t)$, and $[\![(\nabla_1, r_1)]\!] \subseteq [\![(\nabla, t)]\!]$, $[\![(\nabla_1, r_2)]\!] \subseteq [\![(\nabla, t)]\!]$. Then there is an* EQR*-freshness context $\nabla'$, such that $[\![(\nabla', t)]\!] \subseteq [\![(\nabla, t)]\!]$, and $[\![(\nabla_1, r_1)]\!] \subseteq [\![(\nabla', t)]\!]$, $[\![(\nabla_1, r_2)]\!] \subseteq [\![(\nabla', t)]\!]$, and $[\![(\nabla', t)]\!]$ is the minimum that is unique w.r.t. the properties above.*

**Proof.** Two EQR-freshness constraints over the same set $\mathcal{A}$ of atom-variables can be joined as follows: If $(Q_1 \implies P_{1,i}) \wedge \ldots \wedge (Q_n \implies P_{n,i})$ for $i = 1, 2$ are two EQR-freshness constraints over $\mathcal{A}$. Then the join can be represented as the EQR-freshness constraint $(Q_1 \implies (P_{1,1} \wedge P_{1,2})) \wedge \ldots \wedge (Q_n \implies (P_{n,1} \wedge P_{n,2}))$, and the result follows. ◀

▶ **Theorem 4.12.** *Let $(\nabla_{\mathrm{AAU}}, r)$ be a generalization output by the nominal* AtomAntiUnif *algorithm when given $(\nabla, X : r_1 \triangleq r_2)$ as input. Then* AtomAntiUnifLgg *applied to $(\nabla_{\mathrm{AAU}}, r)$ outputs a least general generalization for $(\Delta, X : r_1 \triangleq r_2)$. In addition, this strengthening has a maximal result and it is unique up to equivalences.*

Since the algorithm AtomAntiUnif and the other algorithms also work in the same way if AtomAntiUnif is started with more general terms-in-context where the constraint is an EQR-constraint, we obtain that the nominal anti-unification problem with atom-variables and EQR-freshness constraints is decidable and a minimal complete set of lggs exists and has exactly one generalization.

▶ **Theorem 4.13.** *The nominal anti-unification problem of $\mathtt{NL}_A$-expressions with atom-variables and* EQR*-freshness contexts is decidable and unitary.*

Application of Theorem 3.11 implies that the anti-unification problem for $\mathtt{NL}_A$-expressions and contexts is at most finitary.

▶ **Theorem 4.14.** *Let $A_0$ be a finite set of atom-variables. Then the nominal anti-unification problem of $\mathtt{NL}_A$-expressions and $\mathtt{NL}_A$-freshness contexts and where only atom-variables from $A_0$ may appear in the constraint part of the computed generalizations, is decidable and unitary or finitary.*

▶ **Theorem 4.15.** *If the atom-variables in the constraint parts of the terms-in-context are not restricted, then the nominal anti-unification problem of* $\texttt{NL}_A$ *-expressions and* $\texttt{NL}_A$*-freshness contexts is unitary or finitary.*

Investigating decidability of computing a complete set of lggs in Theorem 4.15 with $\texttt{NL}_A$-freshness contexts and also determining the exact solution type is future research. Applying Theorem 3.11 does not solve the computation problem in the general case of $\texttt{NL}_A$-freshness contexts, since we do not know an upper bound on the number of atom-variables that may occur in the constraint part of the lggs.

## 4.3 The Atoms-Only Case

We reconsider the nominal unification problem treated by Baumgartner et al. [4]. We model this case by adding the global condition that all atom-variables must be instantiated with different atoms and call it the *atoms-only case*. Then simplifications on the expressions of the abstract language are valid, such that only suspensions of generalization variables are needed, and freshness contexts are conjunctions of constraints of the form $A\#X$. Specializing our algorithm is straightforward. The (`Abs`)-rule only needs the condition that a *fresh* atom variable is used for renaming. The four rules (`SusA`), (`SusYY`), (`SolveYY`), (`SolveAB`) now only rely on an easy equality test of permutations. The same for EqvM and EqvBiEx, which now run in polynomial time. If also sharing of the substitution components is obeyed, then AtomAntiUnif runs in polynomial time, and since alternatives do not have to be investigated, one run is sufficient. For computing an lgg, from the result, we extend the algorithm by simply checking for all possible constraints of the form $A\#X$, whether these can be added to the solution and keeping the generalization property. This check is polynomial. The result constraint will be the union of all these constraints that pass the test.

▶ **Theorem 4.16.** *For the atoms-only case, the adapted and extended algorithm* AtomAntiUnif *runs in polynomial time, and always computes a unique lgg of the input* $(\nabla, s)$ *and* $(\nabla, r)$*. This implies that the solution-type is unitary.*

# 5 Conclusion and Future Work

A sound and complete rule-based algorithm (AtomAntiUnif) for nominal anti-unification with atom-variables is provided. Our algorithm relies on an equivariance algorithm (EqvM) that is generalized to atom-variables and nested permutations. It computes, in exponential time, a *unique* (but not necessarily least) generalization of two terms-in-context. We refine the output generalization with the AtomAntiUnifLgg algorithm, which adds EqR-freshness constraints to the solutions, and a unique lgg is obtained. Our approach improves previous results [4], since the nominal anti-unification problem with atom-variables and EqR-freshness constraints is decidable and unitary. The variant with $\texttt{NL}_A$-freshness constraints is unitary or finitary, but its decidability and the determination of the exact solution type is future work. We also describe an improved algorithm of the problem setting of Baumgartner et al.(the atoms-only-case), which is unitary and still requires polynomial time. Future work is to optimize the algorithms, investigate specializations like a restriction to linear generalizations, and to determine the exact complexity of the problem. Applications of nominal techniques in code duplication and recursion scheme detection are also to be investigated. Also an anti-unification algorithm for a core-language of Haskell [20, 12] that takes recursive lets into account (see [22, 23]) would extend applicability.

## References


1 Franz Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. 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 86–97. Springer, 1991. doi:10.1007/3-540-53904-2_88.

2 Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in haskell functions via anti-unification. *Future Gener. Comput. Syst.*, 79:669–686, 2018. doi:10.1016/j.future.2017.07.024.

3 Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014. doi:10.1007/978-3-319-11558-0_38.

4 Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal anti-unification. 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 57–73. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.57.

5 Peter E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-unification algorithms and their applications in program analysis. In Amir Pnueli, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009. doi:10.1007/978-3-642-11486-1_35.

6 Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008. doi:10.1016/j.tcs.2008.05.012.

7 David M. Cerna and Temur Kutsia. Higher-order equational pattern anti-unification. 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 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.FSCD.2018.12.

8 David M. Cerna and Temur Kutsia. Higher-order pattern generalization modulo equational theories. *Math. Struct. Comput. Sci.*, 30(6):627–663, 2020. doi:10.1017/S0960129520000110.

9 James Cheney. Toward a general theory of names: Binding and scope. In *MERLIN 2005*, pages 33–40. ACM, 2005.

10 Hubert Comon. Sufficient completness, term rewriting systems and "anti-unification". In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1986. doi:10.1007/3-540-16780-3_85.

11 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002. doi:10.1007/s001650200016.

12 Haskell. Haskell, an advanced, purely functional programming language, 2019. URL: www.haskell.org.

13 Boris Konev and Temur Kutsia. Anti-unification of concepts in description logic EL. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 227–236. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12880.

14 Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings*, volume 4830 of *Lecture*

*Notes in Computer Science*, pages 273–282. Springer, 2007. doi:10.1007/978-3-540-76928-6_29.

15 Yunus D. K. Kutz and Manfred Schmidt-Schauß. Rewriting with generalized nominal unification. *Math. Struct. Comput. Sci.*, 30(6):710–735, 2020. doi:10.1017/S0960129520000122.

16 Jean-Louis Lassez and Kim Marriott. Explicit representation of terms defined by counter examples. *J. Autom. Reason.*, 3(3):301–317, 1987. doi:10.1007/BF00243794.

17 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *19th RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2008.

18 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl, 2010. doi:10.4230/LIPIcs.RTA.2010.209.

19 Jianguo Lu, John Mylopoulos, Masateru Harao, and Masami Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000. doi:10.1023/A:1018952121991.

20 Simon Marlow, editor. *Haskell 2010 – Language Report*. Haskell.org, 2010.

21 Gordon D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.

22 Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal unification of higher order expressions with recursive let. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *LNCS*, pages 328–344. Springer, 2016. doi:10.1007/978-3-319-63139-4_19.

23 Manfred Schmidt-Schauß, Temur Kutsia, Jordi Levy, Mateu Villaret, and Yunus Kutz. Nominal unification and matching of higher order expressions with recursive let. *Fundamenta Informaticae*, 2022. to be published in volume 185 issue 03.

24 Manfred Schmidt-Schauß, David Sabel, and Yunus D. K. Kutz. Nominal unification with atom-variables. *J. Symb. Comput.*, 90:42–64, 2019. doi:10.1016/j.jsc.2018.04.003.

25 Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In *17th CSL, 12th EACSL, and 8th KGC*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003. doi:10.1007/978-3-540-45220-1_41.

## A Proofs of Section 3 and Section 4

It is convenient to note that given a state $(\Gamma, M, \nabla, \theta)$ obtained from a starting configuration $(\{X : s \triangleq t\}, \emptyset, \nabla, [])$, the instance of the input variable $(X \circ \theta)$ is a context $C[X_1, \ldots, X_n]$ such that $\texttt{GenVar}(M \cup \Gamma) = \{X_1, \ldots, X_n\}$. The AtomAntiUnif rules do not alter the semantics of states in a derivation.

▶ **Lemma A.1.** *Let $(\{X : s \triangleq t\}, \emptyset, \nabla_0, [])$ an input for the* AtomAntiUnif *algorithm and $(\Gamma, M, \nabla, \theta)$ a state in the derivation. If $(\Gamma, M, \nabla, \theta) \Longrightarrow_{(\texttt{R})} (\Gamma', M', \nabla', \theta')$ then $[\![(\Gamma', M', \nabla', \theta')]\!] = [\![(\Gamma, M, \nabla, \theta)]\!]$.*

**Proof.** The proof is by induction on the rule $(\texttt{R})$ from Figure 1 applied in $(\Gamma, M, \nabla, \theta) \Longrightarrow_{(\texttt{R})} (\Gamma', M', \nabla', \theta')$. The interesting case is for rule $(\texttt{Abs})$:

In this case, $\Gamma = \{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\} \cup \Gamma_0$, $\Gamma' = \{X_1 : (W_1\ B) \cdot s \triangleq (W_2\ B) \cdot t\} \cup \Gamma_0$, $M = M'$, $\theta' = \theta \cup \{X_0 \mapsto \lambda B.X_1\}$ and the reduction is: (we assume $\Gamma_0 = M = \emptyset$ w.l.o.g. because they will not be used in this step and to simplify the notation)

$$(\{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta) \Longrightarrow (\{X_1 : (W_1\ B) \cdot s \triangleq (W_2\ B) \cdot t\}, \emptyset, \nabla \cup \nabla', \theta')$$

where $\nabla' = \{B\#\lambda W_1.s, B\#\lambda W_2.t\}$.
$(\supseteq)$ $[\![(\{X_1{:}(W_1\ B) \cdot s \triangleq (W_2\ B) \cdot t\}, \emptyset, \nabla \cup \nabla', \theta')]\!] \subseteq ([\![\{X_0{:}\lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta)]\!]$.

By Definition 3.7 and with $X$ the input variable,

$$
\begin{aligned}
&[\![(\{X_1{:}(W_1\ B)\cdot s \triangleq (W_2\ B)\cdot t\}, \emptyset, \nabla\cup\nabla', \theta')]\!]\\
&= \{[(X\circ\theta')\rho_s] \mid \rho_s \text{ is ground,} (\nabla\cup\nabla')\rho_s \text{ holds}, [X_1\rho_s]_\sim \in [\![(\nabla\cup\nabla', (W_1\ B)\cdot s)]\!]\}\\
&\cup \{[(X\circ\theta')\rho_t] \mid \rho_t \text{ is ground,} (\nabla\cup\nabla')\rho_t \text{ holds}, [X_1\rho_t]_\sim \in [\![(\nabla\cup\nabla', (W_2\ B)\cdot t)]\!]\}
\end{aligned}
$$

By Definition 2.12

$$[\![(\nabla\cup\nabla', (W_1\ B)\cdot s)]\!] = \{[((W_1\ B)\cdot s)\sigma]_\sim \mid \exists\sigma \text{ ground} : (\nabla\cup\nabla')\sigma \text{ holds}\}.$$

Let $\sigma$ be a ground substitution s.t. $(\nabla \cup \nabla')\sigma$ holds, i.e, $\nabla\sigma$ and $\nabla'\sigma = \{B\#\lambda W_1.s, B\#\lambda W_2.t\}\sigma$ hold. Then, $B\sigma\#\lambda W_1\sigma.s\sigma$ and $B\sigma\#\lambda W_2\sigma.t\sigma$. We will analyze some cases:

- $B\sigma = W_1\sigma$.
  Then, $r \sim ((W_1\ B)\cdot s)\sigma = s\sigma$. Thus, $[r]_\sim \in [\![(\nabla\cup\nabla', s)]\!]$ and $[X_1\rho_s]_\sim \in [\![(\nabla\cup\nabla', s)]\!]$.
  Notice that

  $$
  \begin{aligned}
  (X\theta')\rho_s &= (X(\theta\cup\{X_0\mapsto\lambda B.X_1\}))\rho_s\\
  &= C[X_1]\rho_s,\ \text{for an } \mathtt{NL}_A\text{-context } C\\
  &= C'[\lambda B.X_1]\rho_s,\ \text{for } C[] = C'[\lambda B.[]]\\
  &= C'[\lambda B\rho_s.X_1\rho_s] = C'[X_0\rho'_s] = (X\circ\theta)\rho'_s
  \end{aligned}
  $$

  where $[X_0\rho'_s]_\sim \in [\![(\nabla\cup\nabla', \lambda W_1.s)]\!]$. Since $\nabla\rho_s$ holds and

  $$
  \begin{aligned}
  [\![(\{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta)]\!] &= \{[(X\theta)\rho_s] \mid \rho_s \text{ is ground,} \nabla\rho_s \text{ holds}, [X_0\rho_s]_\sim \in [\![(\nabla, \lambda W_1.s)]\!]\}\\
  &\cup \{[(X\theta)\rho_t] \mid \rho_t \text{ is ground,} \nabla\rho_t \text{ holds}, [X_0\rho_t]_\sim \in [\![(\nabla, \lambda W_2.t)]\!]\},
  \end{aligned}
  $$

  one has $[(X\theta')\rho_s]_\sim \in [\![(\{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta)]\!]$, and the result follows.

- $B\sigma \neq W_1\sigma$.
  From $B\sigma\#\lambda W_1\sigma.s\sigma$, it follows that $B\sigma\#s\sigma$.

  - If $W_1$ does not occur in $s$ then $r \sim ((W_1\ B)\cdot s)\sigma = s\sigma$ and the case is similar to the previous.
  - Otherwise, $W_1$ occurs in $s$ and $r \sim ((W_1\sigma\ B\sigma)\cdot s\sigma)$, i.e., we replace all the occurrences of $W_1\sigma$ in $s\sigma$ for $B\sigma$.
    Thus, $[X_1\rho_s]_\sim = [r]_\sim \in [\![(\nabla\cup\nabla', (W_1\ B)\cdot s)]\!]$, for some such $\sigma$, and $(\lambda B.X_1)\rho_s \sim \lambda B\sigma.(W_1\sigma\ B\sigma)\cdot s\sigma$. Then,

    $$
    \begin{aligned}
    (X\circ\theta')\rho_s &= C'[\lambda B.X_1]\rho_s,\ \text{for } C[] = C'[\lambda B.[]]\\
    &= C'[X_0\rho'_s],\ \text{where } [X_0\rho'_s]_\sim \in [\![(\nabla\cup\nabla', \lambda W_1.s)]\!]\\
    &= (X\theta)\rho'_s,\ \text{for some ground } \rho'_s.
    \end{aligned}
    $$

    Then, $[(X\theta')\rho_s]_\sim \in [\![(\{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta)]\!]$, and the result follows.

The proof that $[(X\circ\theta')\rho_t]_\sim \in [\![(\{X_0 : \lambda W_1.s \triangleq \lambda W_2.t\}, \emptyset, \nabla, \theta)]\!]$ is analogous. ◀

▶ **Theorem A.1** (cf.Theorem 3.9). *Given $\mathtt{NL}_A$ expressions $s$ and $t$, and a freshness context $\nabla$. If $(\nabla', r)$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$, then there exists a $\nabla''$ and a derivation $(\{X : s \triangleq t\}, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$ computed by* AtomAntiUnif*, such that $[\![(\Delta\cup\nabla'', X\circ\theta)]\!]$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$, and $[\![(\emptyset, M, \Delta\cup\nabla'', \theta)]\!] \subseteq [\![(\nabla', r)]\!]$.*

**Proof.** The proof is by induction on $r$ by investigating the rules from AtomAntiUnif algorithm (Figure 1) used. We show some cases below.

1. $r = \pi \cdot Y'$ (a suspension of a generalization variable) There are three possible cases:

   a. $s = \pi_1 \cdot Y$, $t = \pi_2 \cdot Y$ and the rule (SusYY) was applied.
   Then,

$$\frac{(\{X : \pi_1 \cdot Y \triangleq \pi_2 \cdot Y\}, \emptyset, \nabla, []) \qquad \nabla \vDash \pi_1 = \pi_2}{(\emptyset, \emptyset, \nabla, \{X \mapsto \pi_1 \cdot Y\})} \text{ (SusYY)}$$

   Since, by hypothesis, $(\nabla', \pi \cdot Y)$ is a generalization of $(\nabla, \pi_1 \cdot Y)$ and $(\nabla, \pi_2 \cdot Y)$, we have that $[\![(\nabla, \pi_1 \cdot Y)]\!] = [\![(\nabla, \pi_2 \cdot Y)]\!] \subseteq [\![(\nabla', \pi \cdot Y')]\!]$.
   By Definition 3.7, $[\![(\emptyset, \emptyset, \nabla, \{X \mapsto \pi_1 \cdot Y\})]\!] = [\![(\nabla, \pi_1 \cdot Y)]\!]$ and the result follows trivially for $\Delta = \nabla$ and $\nabla'' = \emptyset$.

   b. $s : \pi_1 \cdot Y$, $t = \pi_2 \cdot Y$ and the rule (SolveYY) was applied.
   Then $\nabla \nvDash \pi_1 = \pi_2$ and this case follows a similar reasoning used for (SolveAB).

   c. $s$ and $t$ are such that the rule (Solve) was applied.
   Then, $(\{X : s \triangleq t\}, \emptyset, \nabla, []) \Rightarrow_{(\texttt{Solve})} (\emptyset, \{X : s \triangleq t\}, \nabla, [])$ where $Head(s) \neq Head(t)$ and $s, t$ are not both atom-variables. By hypothesis, $(\nabla', \pi \cdot Y')$ is a generalization for $(\nabla, s)$ and $(\nabla, t)$, i.e., $[\![(\nabla, s)]\!] \subseteq [\![(\nabla', \pi \cdot Y')]\!]$ and $[\![(\nabla, t)]\!] \subseteq [\![(\nabla', \pi \cdot Y')]\!]$. By soundness $(\nabla, X)$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$. In addition, $[\![(\emptyset, \{X : s \triangleq t\}, \nabla, [])]\!] = [\![(\nabla, s)]\!] \cup [\![(\nabla, t)]\!] \subseteq [\![(\nabla', \pi \cdot Y')]\!]$. The result follows trivially for $\Delta = \nabla$ and $\nabla'' = \emptyset$.

2. $r = \lambda W.r'$
   Then, $s = \lambda W_1.s'$ and $t = \lambda W_2.t'$, and the following holds:

$$[\![(\nabla, \lambda W_1.s')]\!] \subseteq [\![(\nabla', \lambda W.r')]\!] \quad \text{and} \quad [\![(\nabla, \lambda W_2.t')]\!] \subseteq [\![(\nabla', \lambda W.r')]\!].$$

   In addition, $(\nabla', r')$ is a generalization of $(\nabla, s')$ and $(\nabla, t')$. By induction hypothesis, there exist $\nabla''$ and a derivation $(\{X' : s' \triangleq t'\}, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$ such that $(\Delta \cup \nabla'', X' \circ \theta)$ is a generalization of $(\nabla, s')$ and $(\nabla, t')$, and $[\![(\emptyset, M, \Delta \cup \nabla'', \theta)]\!] \subseteq [\![(\nabla', r')]\!]$.
   Notice that with such $s$ and $t$ we can apply the AtomAntiUnif algorithm as follows

$$(\{X{:}\lambda W_1.s' \triangleq \lambda W_2.t'\}, \emptyset, \nabla, []) \Longrightarrow (\{Y{:}(W_1\ B) \cdot s' \triangleq (W_2\ B) \cdot t'\}, \emptyset, \nabla \cup \nabla_B, \{X \mapsto \lambda B.Y\})$$

   where $Y$ is a fresh variable, $B$ is a fresh atom-variable and $\nabla_B = \{B\#\lambda W_1.s', B\#\lambda W_2.t'\}$. Notice that $s'$ and $(W_1\ B) \cdot s'$ are structurally the same term, with $W_1$ renamed to a fresh atom-variable $B$. The same with $t'$ and $(W_2\ B) \cdot t'$. Then,

$$(\{Y : (W_1\ B) \cdot s' \triangleq (W_2\ B) \cdot t'\}, \emptyset, \nabla \cup \nabla_B, []) \Longrightarrow^* (\emptyset, M', \Delta' \cup \nabla_B, \theta')$$

   where $M', \theta', \Delta'$ are versions of $M, \theta$ and $\Delta$ with $W_1$ and $W_2$ renamed to $B$.

$$\begin{aligned}(\{X{:}\lambda W_1.s' \triangleq \lambda W_2.t'\}, \emptyset, \nabla, []) &\Longrightarrow (\{Y{:}(W_1\ B) \cdot s' \triangleq (W_2\ B) \cdot t'\}, \emptyset, \nabla \cup \nabla_B, \{X \mapsto \lambda B.Y\}) \\ &\Longrightarrow^* (\emptyset, M', \Delta' \cup \nabla_B, \{X \mapsto \lambda B.Y\} \cup \theta')\end{aligned}$$

   Then $(\Delta' \cup \nabla_B, X \circ \theta'')$ is a generalization of $(\nabla, \lambda W_1.s')$ and $(\nabla, \lambda W_2.t')$.

▷ Claim. $[\![(\emptyset, M', \Delta' \cup \nabla_B \cup \nabla'', \theta'')]\!] \subseteq [\![(\nabla', \lambda W.r')]\!]$.

In fact, if $[(X\theta'')\rho]_\sim \in [\![(\emptyset, M', \Delta' \cup \nabla_B \cup \nabla'', \theta'')]\!]$ then $\rho$ is ground, $(\Delta' \cup \nabla_B \cup \nabla'')\rho$ holds, and $\rho$ acts on the variables in the range of $\theta'$ accordingly to $M'$. Notice that $(X\theta'') = (\lambda B.Y)\theta' = \lambda(B\theta').Y\theta'$. Then, $[(X\theta'')\rho]_\sim = [\lambda(B\theta')\rho.(Y\theta')\rho]_\sim$, where $[(Y\theta')\rho]_\sim \in [\![(\emptyset, M', \Delta' \cup \nabla'', \theta')]\!] \subseteq [\![(\nabla', r')]\!]$, by the IH. Therefore, $(X \circ \theta'')\rho \sim \lambda b.r^*$ where $r^* \sim r'\sigma$, for some $\sigma$ ground such that $\nabla'\sigma$ holds, $[(X\theta')\rho]_\sim \in [\![(\nabla', \lambda W.r')]\!]$, and the result follows.

**3.** $r = f(r_1, \ldots, r_n)$.
Then $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ and the following hold: $[\![(\nabla, f(s_1, \ldots, s_n))]\!] \subseteq [\![(\nabla', r)]\!]$ and $[\![(\nabla, f(t_1, \ldots, t_n))]\!] \subseteq [\![(\nabla', r)]\!]$. In addition, $(\nabla', r_i)$ is a generalization of $(\nabla, s_i)$ and $(\nabla, t_i)$ for each $i = 1, \ldots, n$. By the IH, there exist $\nabla_i''$ and derivations $(\{X_i : s_i \triangleq t_i\}, \emptyset, \nabla, []) \Longrightarrow^* (\emptyset, M_i, \Delta_i, \theta_i)$ s.t., for each $i = 1, \ldots, n$, the following hold:
**a.** $(\Delta_i \cup \nabla_i'', X_i\theta_i)$ is a generalization of $(\nabla, s_i)$ and $(\nabla, t_i)$, i.e., $[\![(\nabla, s_i)]\!] \subseteq [\![(\Delta_i \cup \nabla_i'', X_i\theta_i)]\!]$ and $[\![(\nabla, t_i)]\!] \subseteq [\![(\Delta_i \cup \nabla_i'', X_i\theta_i)]\!]$.
**b.** and $[\![(\emptyset, M_i, \Delta_i \cup \nabla_i'', \theta_i)]\!] \subseteq [\![(\nabla', r_i)]\!]$.
Now we need to combine these semantics to obtain the final result. We want to prove that there exists $\nabla''$ and an AtomAntiUnif computation

$$(\{X : f(s_1, \ldots, s_n) \triangleq f(t_1, \ldots, t_n)\}, \emptyset, \nabla, []) \Longrightarrow_{(\mathsf{Dec})} (\Gamma_1, \emptyset, \nabla, \{X \mapsto f(X_1, \ldots, X_n)\}) \Longrightarrow^* (\emptyset, M, \Delta, \theta)$$

where $\Gamma_1 = \{X : f(s_1, \ldots, s_n) \triangleq f(t_1, \ldots, t_n)\}$ and $\theta = \{X \mapsto f(X_1, \ldots, X_n)\} \cup \theta'$ s.t. $(\Delta \cup \nabla'', X \circ \theta)$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$, and $[\![(\emptyset, M, \Delta \cup \nabla'', \theta)]\!] \subseteq [\![(\nabla', r)]\!]$.

▷ Claim. $(\nabla \cup \bigcup_i(\Delta_i \cup \nabla_i''), f(X_1\theta_1, \ldots, X_n\theta_n))$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$.

There are no critical clash among our rules (in applications in the same constraint). Thus, from state $(\{X_1 : s_1 \triangleq t_1, \ldots, X_n : s_n \triangleq t_n\}, \emptyset, \nabla, \{X \mapsto f(X_1, \ldots, X_n)\})$ we can choose to simplify first $X_1$, then $X_2$, then ... until we reach $X_n$.

$$(\{X_1 : s_1 \triangleq t_1, \ldots, X_n : s_n \triangleq t_n\}, \emptyset, \nabla, \{X \mapsto f(X_1, \ldots, X_n)\})$$
$$\vdots$$
$$(\{X_2 : s_2 \triangleq t_2, \ldots, X_n : s_n \triangleq t_n\}, M_1, \nabla \cup \Delta_1, \{X \mapsto f(X_1\theta_1, \ldots, X_n\theta_1)\})$$
$$\vdots$$
$$(\{X_3 : s_3 \triangleq t_3, \ldots, X_n : s_n \triangleq t_n\}, M_1 \cup M_2, \nabla \cup \Delta_1 \cup \Delta_2, \{X \mapsto f(X_1\theta_1, X_2\theta_2 \ldots, X_n\theta_n)\})$$
$$\vdots$$
$$(\emptyset, \bigcup_i M_i, \nabla \cup \bigcup_i \Delta_i, \{X \mapsto f(X_1\theta_1, \ldots, X_n\theta_n)\})$$

From soundness $(\nabla \cup \bigcup_i \Delta_i, f(X_1\theta_1, \ldots, X_n\theta_n))$ is a generalization of $(\nabla, s)$ and $(\nabla, t)$. In addition, $[\![(\nabla \cup \bigcup_i(\Delta_i \cup \nabla_i''), f(X_1\theta_1, \ldots, X_n\theta_n))]\!] \subseteq [\![(\nabla \cup \bigcup_i \Delta_i, f(X_1\theta_1, \ldots, X_n\theta_n))]\!]$. It is straightforward to check that $[\![(\nabla \cup \bigcup_i(\Delta_i \cup \nabla_i''), f(X_1\theta_1, \ldots, X_n\theta_n))]\!]$ is also a generalization of $(\nabla, s)$ and $(\nabla, t)$. We take $\theta := \{X \mapsto f(X_1, \ldots, X_n)\} \cup \theta_1 \cup \ldots \cup \theta_n$, $M = \bigcup_i M_i$, $\Delta = \bigcup_i \Delta_i \cup \nabla$ and $\nabla'' = \bigcup_i \nabla_i''$.

▷ Claim. $[\![(\emptyset, M, \Delta \cup \nabla'', \theta)]\!] \subseteq [\![(\nabla', r)]\!]$.

Let $[(X\theta)\rho]_\sim \in [\![(\emptyset, M, \Delta \cup \nabla'', \theta)]\!]$. Then $\rho$ is ground, $(\Delta \cup \nabla'')\rho$ holds, and $[(X\theta)\rho]_\sim = [(f((X_1\theta_1)\rho, \ldots, (X_n\theta_n)\rho)]_\sim$ where, for each $i$, $[(X_i\theta_i)\rho]_\sim \in [\![(\emptyset, M_i, \Delta_i \cup \nabla_i'', \theta_i)]\!] \subseteq [\![(\nabla', r_i)]\!]$, from item (b) of the IH. Then, $[(f((X_1\theta_1)\rho, \ldots, (X_n\theta_n)\rho)]_\sim \in [\![(\nabla', f(r_1, \ldots, r_n))]\!] = [\![(\nabla', r)]\!]$, and the result follows. ◀

▶ **Proposition A.2** (cf. Proposition 3.8 ). *The algorithm* AtomAntiUnif *never get stuck and will yield a generalization. The number of rule applications is linear.*

**Proof.** The claim can be shown by a measure that is the sum of 2∗(size of $\Gamma$)+ size of $M$, where the permutations are ignored in the size. The steps (`Dec`) and (`Abs`) strictly decrease the size, and are always applicable to the generalization triples of type $(f, f)$, and $\lambda, \lambda)$. (`Solve`) also strictly decreases the size. (`SolveAB`) and (`SusA`) are complementary and remove $W_1 \triangleq W_2$ triples. (`SolveYY`) and (`SusYY`) are also complementary and remove $\pi_1{\cdot}Y \triangleq \pi_2{\cdot}Y$ triples. The other rule is (`Merge`) which removes a triple in $M$. ◀

▶ **Lemma A.3** (cf. Lemma 4.6). *Let $\mathcal{A} = \{A, B\}$ be a set of atom-variables, $X$ and $Y$ be generalization variables. The* EQR- *freshness context* $((A \neq B) \implies A\#X) \wedge ((A = B) \implies A\#Y \wedge B\#Y)$ *cannot be encoded as an* $\mathtt{NL}_A$*-freshness context, i.e, a conjunction of freshness constraints.*

**Proof.** Suppose, by contradiction, that $\{A\#s_1, \ldots, A\#s_n, B\#t_1, \ldots, B\#t_m\}$ is the freshness context that is the encoding of the EQR-freshness context. Then the evaluation mechanism must construct the EQR-freshness context above (up to some modifications for $A = B$).

- If $A = B$, then for every freshness constraint of the form $A\#\lambda\pi \cdot W.s'$, the result will be $A\#\lambda A. \ldots$, since all binders will become equal to $A$, hence the constraint evaluates to `True`. We also derive that the terms $s_i, t_i$ in the constraints that evaluate to the desired freshness constraints $A\#X, B\#Y$ do not contain abstractions.
- Let $C\#\pi{\cdot}Y$ (for $C \in \{A, B\}$) be the constraint that evaluates under $(A = B)$ to $B\#Y$. If $A \neq B$ then evaluating $C\#\pi{\cdot}Y$ will result in either $A\#Y$ or $B\#Y$, which are both not part of the results for $(A \neq B)$ in the generalized freshness context above.

Hence there is no such encoding. ◀

# A Certified Algorithm for AC-Unification

**Mauricio Ayala-Rincón**
Departments of Computer Science and Mathematics, University of Brasília, Brazil

**Maribel Fernández**
Department of Informatics, King's College London, UK

**Gabriel Ferreira Silva**
Department of Computer Science, University of Brasília, Brazil

**Daniele Nantes Sobrinho**
Department of Computing, Imperial College London, UK
Department of Mathematics, University of Brasília, Brazil

## Abstract

Implementing unification modulo Associativity and Commutativity (AC) axioms is crucial in rewrite-based programming and theorem provers. We modify Stickel's seminal AC-unification algorithm to avoid mutual recursion and formalise it in the PVS proof assistant. More precisely, we prove the adjusted algorithm's termination, soundness, and completeness. To do this, we adapted Fages' termination proof, providing a unique elaborated measure that guarantees termination of the modified AC-unification algorithm. This development (to the best of our knowledge) provides the first fully formalised AC-unification algorithm.

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

**Keywords and phrases** AC-Unification, PVS, Certified Algorithms, Formal Methods, Interactive Theorem Proving

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.8

**Related Version** *Extended version available at*: `https://www.mat.unb.br/ayala/`

**Supplementary Material** Source code available through hyperlinks on the paper.

**Funding** Research supported by a FAP-DF (DE 00193.00001175/2021-11) and a CNPq (Universal 409003/2021-2) grant. First author partially funded by a CNPq productivity research grant 313290/2021-0. Fourth author partially funded by Edital DPI/DPG n. 03/2020.

## 1 Introduction

Syntactic unification is the problem of, given terms $s$ and $t$, finding a substitution $\sigma$ such that $\sigma s = \sigma t$. The problem of syntactic unification can be generalised to consider an equational theory $E$. In this case, called $E$-unification, we must find a substitution $\sigma$ such that $\sigma s$ and $\sigma t$ are equal modulo $E$, which we denote $\sigma s \approx_E \sigma t$ [15].

Unification has practical applications in mathematics and computer science. It is used, for instance, in interpreters of logic programming languages such as Prolog, in resolution-based theorem provers, in confluence tests based on critical pairs, and so on [5]. Since associative and commutative operators are frequently used in programming languages and theorem provers, tools to support reasoning modulo **A**ssociativity and **C**ommutativity axioms are often required. The problem of AC-unification has been widely studied in this context (see [22, 5]).

© Mauricio Ayala-Rincón, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes Sobrinho;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 8; pp. 8:1–8:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Related Work.** Unification in the presence of AC-function symbols was first solved by Stickel [21]. He showed how the problem is connected to finding nonnegative integral solutions to linear equations and proved that his algorithm was terminating, sound, and complete for a subclass of the general case [21, 22]. However, Stickel's proof of termination did not apply to the general case: almost a decade after the introduction of this algorithm, Fages proposed a measure fixing the termination proof for the general case [12, 13]. Since then, investigations on solving AC-unification efficiently, on the complexity of AC-unification, and on formalising unification modulo equational theories were carried out.

Regarding solving AC-unification efficiently, Boudet et al. [8] proposed an AC-unification algorithm that explores constraints more efficiently than the standard algorithm. Further, Boudet [7] described and compared an implementation of this algorithm to previous ones. Also, Adi and Kirchner [1] implemented an AC-unification algorithm, proposed benchmarks and showed that their algorithm improves over previous ones in time and space.

Regarding the complexity of AC-unification, Benanav et al. [6] showed that the decision problem for AC-matching is NP-complete, and the decision problem for AC-unification is NP-hard. In addition, Kapur and Narendran [16] showed that the complexity of computing a complete set of AC-unifiers is double-exponential.

As far as we know, there are no formalisations of AC-unification algorithms. Nevertheless, there are formalisations of related algorithms, and some preliminary work has been done.

Ayala-Rincón et al. [2] formalised nominal $\alpha$-equivalence for associative, commutative and associative-commutative function symbols. That work is in the nominal setting (see [20]), which encompasses first-order AC-equivalence.

In 2004, Contejean [11] gave a certified AC-matching algorithm in Coq. AC-matching is an easier problem (see Remark 8) related with AC-unification, where we must find a substitution $\sigma$ such that $\sigma s \approx_{AC} t$. A formalisation of nominal C-unification, which can also handle nominal C-matching, is also available [3]. Additionally, Meßner et al. [17] gave a formally verified solver for homogeneous linear Diophantine equations in Isabelle/HOL. As we shall see, the problem of AC-unification is connected to solving linear Diophantine equations.

It is well-known that although both C- and AC-unification problems are of finitary type, the complexity of computing a complete set of unifiers for the former problem is exponential, while for the latter one, it is double-exponential [16]. Indeed, to build minimal complete sets of C-unifiers, only simple swapping-argument-combinations need to be considered to instantiate variables. However, to build minimal complete sets of AC-unifiers, all possible associations and permutations of arguments should be considered, which is precisely expressed by Stickel's method based on solving Diophantine equations.

**Contribution and Applications.** In this work, we give the first (as far as we know) formalisation of termination, soundness and completeness of an algorithm for AC-unification. We formalised Stickel's algorithm for AC-unification using the proof assistant PVS [18]. We chose PVS since we want, as future work (see Section 5), to enrich the nominal unification library that already exists in PVS with a nominal AC-unification algorithm.

When deciding which AC-unification algorithm to formalise, we looked for concise and well-established algorithms, which led us to select Stickel's algorithm, using Fages' proof of termination. We apply minor modifications to Stickel's AC-unification algorithm in order to avoid mutual recursion (PVS does not allow mutual recursion directly, although this can be emulated using PVS higher-order features, see [19]) and to ease the formalisation.

Our formalisation could be used as a starting point to prove the correctness of more efficient algorithms. For instance, when we solve the linear Diophantine equations necessary for AC-unification, we do it until a certain bound is reached, proved sufficient by Stickel [22]. One possible way to sharpen our formalisation is to use a smaller bound, such as the one mentioned by Clausen and Fortenbacher [10]. Another possible way to improve the efficiency of the algorithm is to solve the mentioned Diophantine equations more efficiently, using the graph approach, also described in [10]. Adapting our formalisation to algorithms that use directed acyclic graphs (DAGs) to represent terms (e.g., Boudet's [7]) would imply a reformulation of almost all subtheories of the formalisation due to their dependency on terms. But such a reformulation would be possible and faster than starting from scratch as discussed in Remark 36, Appendix B.

**Organisation.** Section 2 gives the necessary background; Section 3 explains the modification of Stickel's algorithm; Section 4 discusses the most interesting points of the formalisation; finally, Section 5 concludes and discusses possible paths of future work. The appendices provide further details about the algorithms, the PVS code and the proofs. In addition to the appendices, we include cyan-coloured hyperlinks to specific points of interest of the PVS formalisation.

## 2 Background and Example

From now on, we omit the subscript and write that $t$ and $s$ are equal modulo AC as $t \approx s$.

▶ **Definition 1** (Terms). *Let $\Sigma$ be a signature with function symbols and AC-function symbols. Let $\mathcal{X}$ be a set of variables. The set $T(\Sigma, \mathcal{X})$ is generated by the grammar:*

$$s, t ::= a \mid X \mid \langle\rangle \mid \langle s, t\rangle \mid f\ t \mid f^{AC}\ t$$

*where $a$ denotes a constant, $X$ a variable, $\langle\rangle$ is the unit, $\langle s, t\rangle$ is a pair, $f\ t$ is a function application and $f^{AC} t$ is an associative-commutative function application.*

Terms were specified as shown in Definition 1 to make it easier to eventually adapt the formalisation to the nominal setting in future work. That is the reason why the unit (an element in the grammar of the nominal terms) appears in Definition 1. Pairs are used to represent tuples with an arbitrary number of terms. For instance, the pair $\langle t_1, \langle t_2, t_3\rangle\rangle$ represents the tuple $(t_1, t_2, t_3)$. In Definition 1 we imposed that a function application is of the form $ft$, which is not a limitation since $t$ can be a pair. For instance, the term $f(a, b, c)$ can be represented as $f\langle\langle a, b\rangle, c\rangle$ and its arguments are $a$, $b$ and $c$.

▶ **Definition 2** (Well-formed Terms). *We say that a term $t$ is well-formed if $t$ is not a pair and every AC-function application that is a subterm of $t$ has at least two arguments.*

▶ **Definition 3** (AC-Unification problem). *An AC-unification problem is a finite set of equations $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. The left-hand side of the unification problem $P$ is defined as $\{t_1, \ldots, t_n\}$ while the right-hand side is defined as $\{s_1, \ldots, s_n\}$.*

▶ **Notation 1** (AC-Unification pairs). *When $t$ and $s$ are both* headed by the same AC-function symbol*, we refer to the equation $t \approx^? s$ as an AC-unification pair.*

To ease our formalisation (more details in the extended version), we have restricted the terms in the unification problem that our algorithm receives to well-formed terms. Excluding pairs is natural since they are used to encode (lists of) arguments of functions.

▶ **Notation 2.** *When convenient, we may mention that a function symbol $f$ is an AC-function symbol, omit the superscript and write simply $f$ instead of $f^{AC}$.*

▶ **Notation 3** (Flattened form of AC-functions). *When convenient, we may denote in this paper an AC-function in flattened form. For instance, the term $f^{AC}\langle f^{AC}\langle a,b\rangle, f^{AC}\langle c,d\rangle\rangle$ may be denoted simply as $f^{AC}(a,b,c,d)$. In our formalisation (for instance in function $Args_f$), when we manipulate an AC-function term $t$ we are more interested in its arguments than in how they were encoded using pairs.*

▶ **Notation 4** ($Vars$). *We denote the set of variables of a term $t$ by $Vars(t)$. Similarly, we denote the set of variables that occur in a unification problem $P$ as $Vars(P)$.*

A substitution $\sigma$ is a function from variables to terms, such that $\sigma X \neq X$ only for a finite set of variables, called the domain of $\sigma$ and denoted as $dom(\sigma)$. The image of $\sigma$ is then defined as $im(\sigma) = \{\sigma X \mid X \in dom(\sigma)\}$. A well-formed substitution only instantiates variables to well-formed terms. In the proofs of soundness and completeness of the algorithm, we restrict ourselves to well-formed substitutions. Let $V$ be a set of variables. If $dom(\sigma) \subseteq V$ and $Vars(im(\sigma)) \subseteq V$ we write $\sigma \subseteq V$. In our PVS code, substitutions are represented by a list, where each entry of the list is called a nuclear substitution and is of the form $\{X \to t\}$.

▶ **Definition 4** (Nuclear substitution action on terms). *A nuclear substitution $\{X \to s\}$ acts over a term by induction as shown below:*

- $\{X \to s\}a = a$
- $\{X \to s\}\langle\rangle = \langle\rangle$
- $\{X \to s\}Y = \begin{cases} s & \textit{if } X = Y \\ Y & \textit{otherwise} \end{cases}$
- $\{X \to s\}\langle t_1, t_2\rangle = \langle \{X \to s\}t_1, \{X \to s\}t_2\rangle$
- $\{X \to s\}(f\ t_1) = f\ (\{X \to s\}t_1)$
- $\{X \to s\}(f^{AC}\ t_1) = f\ (\{X \to s\}t_1)$

▶ **Definition 5** (Substitution acting on terms). *Since a substitution $\sigma$ is a list of nuclear substitutions, the action of a substitution is defined as:*

- *$nil\ t = t$, where $nil$ is the null list, used to represent the identity substitution*
- *$\textsc{cons}(\{X \to s\},\ \sigma)\ t = \{X \to s\}(\sigma t)$*

▶ Remark 6. Notice that in the definition of action of substitutions the nuclear substitution in the head of the list is applied last. This allows us to, given substitutions $\sigma$ and $\delta$, obtain the substitution $\sigma \circ \delta$ in our code simply as $\textsc{append}(\sigma, \delta)$.

▶ **Notation 5.** *From now on, when composing two substitutions $\sigma$ and $\delta$ we may omit the composition symbol and write $\sigma\delta$ instead of $\sigma \circ \delta$.*

We now define AC-unification unifiers and complete set of unifiers (Definition 7).

▶ **Definition 7** (AC-unifiers). *Let $P$ be a unification problem $\{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. An AC-unifier or solution of $P$ is a substitution $\sigma$ such that $\sigma t_i \approx \sigma s_i$ for every $i$ from 1 to $n$.*

*A substitution $\sigma$ is* more general (modulo AC) than *a substitution $\sigma'$ in a set of variables $V$ if there is a substitution $\delta$ such that $\sigma' X \approx \delta\sigma X$, for all variables $X \in V$. In this case we write $\sigma \leq_V \sigma'$. When $V$ is the set of all variables, we write $\sigma \leq \sigma'$.*

*With the notion of more general substitution, we can define a complete set $\mathcal{C}$ of unifiers of $P$ as a set that satisfies two conditions: each $\sigma \in \mathcal{C}$ is an AC-unifier of $P$; and for every $\delta$ that unifies $P$, there is $\sigma \in \mathcal{C}$ such that $\sigma \leq_{Vars(P)} \delta$.*

We represent an AC-unification problem $P$ as a list in our PVS code, where each element of the list is a pair $(t_i, s_i)$ that represents an equation $t_i \approx^? s_i$. Finally, given a unification problem $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$, we define $\sigma P$ as $\{\sigma t_1 \approx^? \sigma s_1, \ldots, \sigma t_n \approx^? \sigma s_n\}$.

## 2.1 What Makes AC-unification Hard

Let $f$ be an associative-commutative function symbol. Finding a complete set of unifiers for $\{f(X_1, X_2) \approx^? f(a, Y)\}$ is not as easy as it appears at first sight, since it is not enough to simply compare the arguments of the first term with the arguments of the second term. Indeed, this strategy would give us only $\sigma_1 = \{X_1 \to a, Y \to X_2\}$ and $\sigma_2 = \{X_2 \to a, Y \to X_1\}$ as solutions, missing for example the substitution $\sigma_3 = \{X_1 \to f\langle a, W\rangle, Y \to f\langle X_2, W\rangle\}$. This solution would be missed because the arguments of $\sigma_3 Y = f\langle X_2, W\rangle$ are partially contained in $\sigma_3 X_1 = f\langle a, W\rangle$ and partially contained in $\sigma_3 X_2 = X_2$.

▶ Remark 8. In contrast to AC-unification, to guarantee the completeness of AC-matching, it is enough to explore all possible pairings of the arguments of the first term with the arguments of the second term. Evidence of the difficulty of AC-unification is the fact that, although Contejean formalised AC-matching in 2004 (see [11]), until now, there has been no formalisation of AC-unification.

## 2.2 An Example

Before presenting the pseudocode for the algorithm we formalised, we give a higher-level example (taken from the very accessible [22]) of how we would solve $\{f(X, X, Y, a, b, c) \approx^? f(b, b, b, c, Z)\}$. In a high-level view, this technique converts an AC-unification problem into a linear Diophantine equation and uses a basis of solutions of the Diophantine equation to get a complete set of AC-unifiers to our original problem.

The first step is to eliminate common arguments in the terms that we are trying to unify. The problem is now $\{f(X, X, Y, a) \approx^? f(b, b, Z)\}$. The second step is to associate our unification problem with a linear Diophantine equation, where each argument of our terms corresponds to one variable in the equation (this process is called variable abstraction) and the coefficient of this variable in the equation is the number of occurrences of the argument. In our case, the linear Diophantine equation obtained is: $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$ (variable $X_1$ was associated with argument $X$, variable $X_2$ with the argument $Y$ and so on; the coefficient of variable $X_1$ is two, since argument $X$ occurs twice in $f(X, X, Y, a)$ and so on).

**Table 1** Solutions for the equation $2X_1 + X_2 + X_3 = 2Y_1 + Y_2$.

| $X_1$ | $X_2$ | $X_3$ | $Y_1$ | $Y_2$ | New Vars. |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | $Z_1$ |
| 0 | 1 | 0 | 0 | 1 | $Z_2$ |
| 0 | 0 | 2 | 1 | 0 | $Z_3$ |
| 0 | 1 | 1 | 1 | 0 | $Z_4$ |
| 0 | 2 | 0 | 1 | 0 | $Z_5$ |
| 1 | 0 | 0 | 0 | 2 | $Z_6$ |
| 1 | 0 | 0 | 1 | 0 | $Z_7$ |

The third step is to generate a basis of solutions to the equation and associate a new variable (the $Z_i$s) to each solution. As we shall soon see, the unification problem $\{f(X, X, Y, a) \approx^? f(b, b, Z)\}$ may branch into (possibly) many unification problems and the new variables $Z_i$s will be the building blocks for the right-hand side of these unification problems. The result is shown on Table 1. Observing Table 1 we relate the "old variables"

($X_i$s and $Y_i$s) with the "new variables" ($Z_i$s):

$$\begin{aligned} X_1 &= Z_6 + Z_7 \\ X_2 &= Z_2 + Z_4 + 2Z_5 \\ X_3 &= Z_1 + 2Z_3 + Z_4 \\ Y_1 &= Z_3 + Z_4 + Z_5 + Z_7 \\ Y_2 &= Z_1 + Z_2 + 2Z_6. \end{aligned} \tag{1}$$

In order to explore all possible solutions, we must consider whether we will include or not each solution on our basis. Since seven solutions compose our basis (one for each variable $Z_i$), this means that *a priori* there are $2^7$ cases to consider. Considering that including a solution of our basis means setting the corresponding variable $Z_i$ to 1 and not including it means setting it to 0, we must respect the constraint that no original variables $(X_1, X_2, X_3, Y_1, Y_2)$ receive 0. Eliminating the cases that do not respect this constraint, we are left with 69 cases.

For example, if we decide to include only the solutions represented by the variables $Z_1$, $Z_4$ and $Z_6$, the corresponding unification problem, according Equations (1), becomes:

$$P = \{X_1 \approx^? Z_6, X_2 \approx^? Z_4, X_3 \approx^? f(Z_1, Z_4), Y_1 \approx^? Z_4, Y_2 \approx^? f(Z_1, Z_6, Z_6)\}. \tag{2}$$

We can also drop the cases where a variable that does not represent a variable term is paired with an AC-function application. For instance, the unification problem $P$ should be discarded, since the variable $X_3$ represents the constant $a$, and we cannot unify $a$ with $f(Z_1, Z_4)$. This constraint eliminates 63 of the 69 potential unifiers.

Finally we replace the variables $X_1, X_2, X_3, Y_1, Y_2$ by the original arguments they substituted and proceed with the unification. Some unification problems that we will explore will be unsolvable and discarded later, as: $\{X \approx^? Z_6, Y \approx^? Z_4, a \approx^? Z_4, b \approx^? Z_4,\ Z \approx^? f(Z_6, Z_6)\}$ (we cannot unify both $a$ with $Z_4$ and $b$ with $Z_4$ simultaneously). In the end, the solutions computed will be:

$$\begin{aligned} &\sigma_1 = \{Y \to f(b, b), Z \to f(a, X, X)\}, && \sigma_2 = \{Y \to f(Z_2, b, b), Z \to f(a, Z_2, X, X)\}, \\ &\sigma_3 = \{X \to b, Z \to f(a, Y)\}, && \sigma_4 = \{X \to f(Z_6, b), Z \to f(a, Y, Z_6, Z_6)\}. \end{aligned} \tag{3}$$

▶ Remark 9. When using the technique described in this section to unify $f(X, X, Y, a, b, c)$ with $f(b, b, b, c, Z)$, we obtained unification problems that only contain the variables $X_1$, $X_2$, $X_3$, $Y_1$, $Y_2$ or AC-functions whose arguments are all variables (for instance $P$ in Equation 2). However, this does not mean that our technique cannot be applied to general AC-unification problems, since we eventually replace the variables $X_1, X_2, X_3, Y_1, Y_2$ by their corresponding arguments $(X, Y, a, b, Z$ respectively) and proceed with unification.

▶ Remark 10 (Cases on AC1-Unification). If we were considering AC1-unification, where our signature has an identity `id` function symbol, we could consider only the case where we include all the AC solutions in our basis and instantiate the variables $Z_i$s later on to be `id`.

## 3 Algorithm

For readability, we present the pseudocode of the algorithms, instead of the actual PVS code. We have formalised Algorithm 1 to be terminating, sound and complete. Moreover, the algorithm is functional and keeps track of the current unification problem $P$, the substitution $\sigma$ computed so far, and the variables $V$ that are/were in the problem. The output is a list of

**Algorithm 1** Algorithm to Solve an AC-Unification Problem $P$.

```
1: procedure ACUnif(P, σ, V)
2:   if nil?(P) then return cons(σ, NIL)
3:   else let ((t, s), P1) = CHOOSE(P) in
4:     if (s matches X) and (X not in t) then
5:       σ1 = {X → t}
6:       return ACUnif(σ1 P1, APPEND(σ1, σ), V)
7:     else
8:       if t matches a then
9:         if s matches a then return ACUnif(P1, σ, V)
10:        else return NIL
11:      else if t matches X then
12:        if X not in s then
13:          σ1 = {X → s}
14:          return ACUnif(σ1 P1, APPEND(σ1, σ), V)
15:        else if s matches X then return ACUnif(P1, σ, V)
16:        else return NIL
17:      else if t matches ⟨⟩ then
18:        if s matches ⟨⟩ then return ACUnif(P1, σ, V)
19:        else return NIL
20:      else if t matches f t1 then
21:        if s matches f s1 then
22:          (P2, bool) = DECOMPOSE(t1, s1)
23:          if bool then return ACUnif(APPEND(P2, P1), σ, V)
24:          else return NIL
25:        else return NIL
26:      else
27:        if s matches f^AC s1 then
28:          InputLst = APPLYACStep(P, NIL, σ, V)
29:          LstResults = MAP(ACUnif, InputLst)
30:          return FLATTEN (LstResults)
31:        else return NIL
```

substitutions, where each substitution $\delta$ in this list is an AC-unifier of $P$. The first call to the algorithm, in order to unify two terms $t$ and $s$, is done with $P = cons((t, s), nil)$, $\sigma = nil$ (because we have not computed any substitution yet) and $V = Vars((t, s))$.

The algorithm explores the structure of terms. It starts by analysing the list $P$ of terms to unify. If it is empty (line 2), we have finished, and the algorithm returns a list containing only one element: the substitution $\sigma$ computed so far. Otherwise the algorithm calls the auxiliary function CHOOSE (line 3), that returns a pair $(t, s)$ and a unification problem $P_1$, such that $P = \{t \approx^? s\} \cup P_1$. The algorithm will try to simplify our unification problem $P$ by simplifying $\{t \approx^? s\}$, and it does that by seeing what the form of $t$ and $s$ is.

▶ Remark 11. The algorithm does not check arity consistency of the input.

## 3.1 The Functions choose and decompose

The function CHOOSE selects a unification pair from the input problem, avoiding AC-unification pairs if possible. This means that we will only enter on the **if** of line 27 of ACUnif (see Algorithm 1) when $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$ is such that for every $i$,

$t_i \approx^? s_i$ is an AC-unification pair. This heuristic aid us in the proof of termination; makes the algorithm more efficient, since it guarantees that we only enter on the AC-part of the algorithm when we need it (the AC-part is the computationally heaviest); and is not a significant deviation from Stickel's algorithm [22].

If the function DECOMPOSE receives two terms $t$ and $s$ and these terms are both pairs, it recursively tries to decompose them, returning a tuple $(P, bool)$, where $P$ is a unification problem and *bool* is a boolean that is $True$ if the decomposition was successful. If neither $t$ nor $s$ is a pair, the unification problem returned is just $P = \{t \approx^? s\}$ and $bool = True$. If one of the terms is a pair and the other is not, the function returns (NIL, $False$). In Algorithm 1, we call DECOMPOSE ($t_1$, $s_1$) when we encounter an equation of the form $ft_1 \approx^? fs_1$ and therefore guarantee that all the terms in the unification problem remain well-formed. Although it would have been correct to simplify an equation of the form $ft_1 \approx^? fs_1$ to $t_1 \approx^? s_1$, if $t_1$ or $s_1$ were pairs we would not respect our restriction that only well-formed terms are in our unification problem.

▶ **Example 12.** Below we give examples of function DECOMPOSE.

- DECOMPOSE$(\langle a, \langle b, c\rangle\rangle,\ \langle c, \langle X, Y\rangle\rangle) = (\{a \approx^? c,\ b \approx^? X,\ c \approx^? Y\},\ True)$
- DECOMPOSE$(a, Y) = (\{a \approx^? Y\},\ True)$
- DECOMPOSE$(X, \langle c, d\rangle) =$ (NIL, $False$)

## 3.2 The AC-part of the Algorithm

The AC-part of Algorithm 1 relies on function APPLYACSTEP (Section 3.2.4), which depends on two functions: SOLVEAC (Section 3.2.1) and INSTANTIATESTEP (Section 3.2.3). Since there are multiple possibilities for simplifying each AC-unification pair, APPLYACSTEP will return a list (*InputLst* in Algorithm 1), where each entry of the list corresponds to a branch Algorithm 1 will explore (line 28). Each entry in the list is a triple that will be given as input to ACUNIF, where the first component is the new AC-unification problem, the second component is the substitution computed so far and the third component is the new set of variables that are/were in use. After ACUNIF calls APPLYACSTEP, it explores every branch generated by calling itself recursively on every input in *InputLst* (line 29 of Algorithm 1). The result of calling MAP(ACUNIF, *InputLst*) is a list of lists of substitutions. This result is then flattened into a list of substitutions and returned.

### 3.2.1 Function solveAC

The function SOLVEAC does what was illustrated in the example of Section 2.2. While APPLYACSTEP or ACUNIF take as part of the input the whole unification problem, SOLVEAC takes only two terms $t$ and $s$. It assumes that both terms are headed by the same AC-function symbol $f$. It also receives as input the set of variables $V$ that are/were in the problem (since SOLVEAC will introduce new variables, we must know the ones that are/were already in use).

The first step is to eliminate common arguments of both $t$ and $s$. This is done by function ELIMCOMARG, which returns the remaining arguments and their multiplicity.

To ease the formalisation we do not calculate a basis of solutions for the linear Diophantine equation, but a spanning set (which is not necessarily linearly independent). To generate this spanning set, it suffices to calculate all the solutions until an upper bound, computed by function CALCULATEUPPERBOUND. Given a linear Diophantine equation $a_1X_1 + \ldots + a_mX_m = b_1Y_1 + \ldots + b_nY_n$, our upper bound (taken from [21]) is the maximum of $m$ and

$n$ times the maximum of all the least common multiples ($lcm$) obtained by pairing each one of the $a_i$s with each one of the $b_j$s. In other words, our upper bound is: $max(m, n) * max_{i,j}(lcm(a_i, b_j))$.

$$D = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

The function DIOSOLVER receives as input the multiplicity of the arguments of $t$ and $s$ and the upper bound calculated by CALCULATEUPPERBOUND and calculates the spanning set of solutions, returning a matrix. For instance, the Table 1 of the Example in Section 2.2 would be represented in our code as the matrix $D$. Each row of $D$ is associated with one solution and thus with one of the new variables. Each column of $D$ is associated with one of the arguments of $t$ or $s$. Modifying DIOSOLVER to calculate a basis of solutions (for instance, by using the method described in [10]) instead of a spanning set would certainly improve the efficiency of the algorithm.

To explore all possible cases, we must decide whether or not we will include each solution. In our code, this translates to considering submatrices of $D$ by eliminating some rows. In the example of Section 2.2, we mentioned that we should observe two constraints:

- no "original variable" (the variables $X_1, \ldots, X_m, Y_1, \ldots, Y_n$ associated with the arguments of $t$ and $s$) should receive the value 0. In terms of $D$, it means every column has at least one coefficient different than zero.
- an original variable, which does not represent a variable term, cannot be paired with an AC-function application. In terms of $D$, it means that a column corresponding to one non-variable argument has one coefficient equal to 1 and all the remaining coefficients equal to 0.

The function in our PVS code that extracts (a list of) the submatrices of $D$ that satisfies these constraints is EXTRACTSUBMATRICES. Let *SubmatrixLst* be this list.

Finally, we translate each submatrix $D_1$ in *SubmatrixLst* into a new unification problem $P_1$, by calling function DIOMATRIX2ACSOL. For instance, the unification problem $P_1 = \{X \approx^? Z_6, Y \approx^? Z_4, a \approx^? Z_4, b \approx^? Z_4, Z \approx^? f(Z_6, Z_6)\}$ would be obtained from submatrix $D_1$.

$$D_1 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Notice that this is the submatrix associated with a solution including only the rows 4 and 6 (of the variables $Z_4, Z_6$).

The function DIOMATRIX2ACSOL also updates the variables that are/were in the unification problem, to include the new variables $Z_i$s introduced. In our example, the new set of variables that are/were in the problem is $V_1 = \{X, Y, Z, Z_4, Z_6\}$. Therefore, the output of DIOMATRIX2ACSOL is a pair, where the first component is the new unification problem (in our example $P_1$) and the second component is the new set of variables that are/were in use (in our example $V_1$). The output of SOLVEAC is the list of pairs obtained by applying DIOMATRIX2ACSOL to every submatrix in *SubmatrixLst*.

### 3.2.2 Common Structure of Unification Problems Returned by solveAC

Suppose function SOLVEAC receives as input the terms $u$ and $v$, both headed by the same AC-function symbol $f$. Let $u_1, \ldots, u_m$ be the different arguments of $u$ and let $v_1, \ldots, v_n$ be the different arguments of $v$, after eliminating the common arguments of $u$ and $v$. If $P_1 = \{t_1 \approx^? s_1, \ldots, t_k \approx^? s_k\}$ is one of the unification problems generated by function SOLVEAC, when it receives as input $u$ and $v$ then:

1. $k = m + n$ and the left-hand side of this unification problem (i.e., the terms $t_1, \ldots, t_k$) are the different arguments of $u$ and $v$:

$$t_i = \begin{cases} u_i, & \text{if } i \leq m \\ v_{i-m} & \text{otherwise.} \end{cases}$$

2. The terms in the right-hand side of this problem (i.e., the terms $s_1, \ldots, s_k$) are introduced by SOLVEAC and are either new variables $Z_i$s or AC-functions headed by $f$ whose arguments are all new variables $Z_i$s (This is how we obtained the problem in (2)).
3. A term $s_i$ is an AC-function headed by $f$ only if the corresponding term $t_i$ is a variable.

### 3.2.3 Function instantiateStep

After the application of function SOLVEAC, we instantiate the variables that we can by calling function INSTANTIATESTEP. Indeed, for the proof of termination, it is necessary to compose the substeps of the algorithm with some strategy, as the following example (adapted from [13]) shows.

▶ **Example 13** (Looping forever). Let $f$ be an AC-function symbol. Suppose we want to solve $P = \{f(X,Y) \approx^? f(U,V), X \approx^? Y, U \approx^? V\}$ and instead of instantiating the variables as soon as we can, we decide to try solving the first equation. Applying function SOLVEAC to try to unify $f(X,Y)$ with $f(U,V)$ we obtain as one of the branches the unification problem $\{X \approx^? f(X_1,X_2), Y \approx^? f(X_3,X_4), U \approx^? f(X_1,X_3), V \approx^? f(X_2,X_4)\}$. We can solve this branch by instantiating $X$, $Y$, $U$ and $V$. After these instantiations, we have to unify the remaining two equations: $\{f(X_1,X_2) \approx^? f(X_3,X_4), f(X_1,X_3) \approx^? f(X_2,X_4)\}$. Solving the first equation, one branch obtained is $\{X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which get us back to $P' = \{f(X_1,X_3) \approx^? f(X_2,X_4), X_1 \approx^? X_3, X_2 \approx^? X_4\}$, which is essentially the same unification problem we started with.

This infinite loop in our example would not have happened if we had instantiated $\{X \to Y\}$ and $\{U \to V\}$ in the beginning. To prevent this from happening, Algorithm 1 only handles AC-unification pairs when there are no equations $s \approx^? t$ of other type left, and as soon as we apply the function SOLVEAC we immediately instantiate the variables that we can by calling function INSTANTIATESTEP.

### 3.2.4 Function applyACStep

Function APPLYACSTEP relies on functions SOLVEAC and INSTANTIATESTEP, and is called by Algorithm 1 when all the equations $s \approx^? t \in P$ are AC-unification pairs. In a very high-level view, it applies functions SOLVEAC and INSTANTIATESTEP to every AC-unification pair in the unification problem $P$. It receives as input a unification problem, which is partitioned in sets $P_1$ and $P_2$, a substitution $\sigma$, and the set of variables to avoid $V$. $P_1$ and $P_2$ are, respectively, the subset of the unification problem for which functions SOLVEAC and

INSTANTIATESTEP have not been called, and the subset to which we have already called these functions. The substitution $\sigma$ is the substitution computed so far. Therefore, the first call to this function is with $P_2 = nil$ and as the function goes recursively calling itself, $P_1$ diminishes while $P_2$ increases.

# 4 Interesting Points on the Formalisation

## 4.1 Avoiding Mutual Recursion

When specifying Stickel's algorithm, we tried to follow closely the pseudocode presented in [13] (the papers [21, 22] give a higher-level description of the algorithm). In [13] there is a function UNIAC used to unify terms $t$ and $s$ and a function UNICOMPOUND used to unify a list of terms $(t_1, \ldots, t_n)$ with a list of terms $(s_1, \ldots, s_n)$. These functions are mutually recursive, i.e. UNIAC calls UNICOMPOUND and vice-versa, something not allowed in PVS[1] [19].

We have adapted the algorithm to use only one main function, which receives a unification problem $P$ and operates (except for the AC-part of the algorithm, see Section 3.2) by simplifying one of the equations $\{t \approx^? s\}$ of $P$. The main modification is that the lexicographic measure we use (adapted from [13]) would not diminish if in the AC-part of the unification problem we had simplified only one of the equations $\{t \approx^? s\}$ of $P$ (see the discussion in Section 4.3.2).

## 4.2 The Lexicographic Measure

To prove termination in PVS, we must define a measure and show that this measure decreases at each recursive call the algorithm makes. We have chosen a lexicographic measure with four components: $lex = (|V_{NAC}(P)|,\ |V_{>1}(P)|,\ |AS(P)|,\ size(P))$, where $V_{NAC}(P)$, $V_{>1}(P)$, $AS(P)$, $size(P)$ are given in Definitions 14, 18, 21 and 23, respectively. Table 2 shows which components do not increase (represented by $\leq$) and which components strictly decrease (represented by $<$) for each recursive call that Algorithm 1 makes.

▶ **Definition 14** ($V_{NAC}(P)$). *We denote by $V_{NAC}(P)$ the set of variables that occur in the problem $P$ excluding those that only occur as arguments of AC-function symbols.*

▶ **Example 15.** Let $f$ be an AC-function symbol and let $g$ be a standard function symbol. Let $P = \{X \approx^? a, f(X, Y, W, g(Y)) \approx^? Z\}$. Then $V_{NAC}(P) = \{X, Y, Z\}$.

Before defining $V_{>1}(P)$, we need to define the subterms of a unification problem.

▶ **Definition 16** ($Subterms(P)$). *The subterms of a unification problem $P$ are given as: $Subterms(P) = \bigcup_{t \in P} Subterms(t)$, where the notion of subterms of a term $t$ excludes all pairs and is defined recursively as follows:*

- $Subterms(a) = \{a\}$
- $Subterms(Y) = \{Y\}$
- $Subterms(\langle\rangle) = \{\langle\rangle\}$
- $Subterms(\langle t_1, t_2\rangle) = Subterms(t_1) \cup Subterms(t_2)$
- $Subterms(f\ t_1) = \{f\ t_1\} \cup Subterms(t_1)$
- $Subterms(f^{AC} t_1) = \bigcup_{t_i \in Args(f^{AC} t_1)} Subterms(t_i) \cup \{f^{AC} t_1\}$

*Here, $Args(f^{AC} t_1)$ denote the arguments of $f^{AC}\ t_1$.*

[1] Despite this restriction, since PVS has higher-order logic foundations, mutual recursion can be emulated, as usual, using functional parameters. However, this would imply a treatment of such parameter functions that restricts their domains according to the chosen measure.

▶ Remark 17 (Subterms of AC and non-AC functions). The definition of subterms for non-AC functions cannot be used for AC functions, as the following counterexample shows. Let $f$ be an AC-function symbol and consider the term $t = f\langle f\langle a, b\rangle, f\langle c, d\rangle\rangle$. Then $Subterms(t) = \{t, a, b, c, d\}$. However, if we had used the definition of subterms for non-AC functions, we would obtain $Subterms(t) = \{t, f\langle a, b\rangle, f\langle c, d\rangle, a, b, c, d\}$.

▶ **Definition 18** ($V_{>1}(P)$). *We denote by $V_{>1}(P)$ the set of variables that are arguments of (at least) two terms $t$ and $s$ such that $t$ and $s$ are headed by different function symbols and $t$ and $s$ are in $Subterms(P)$. The informal meaning is that if $X \in V_{>1}(P)$ then $X$ is an argument to at least two different function symbols.*

▶ **Example 19.** Let $f$ be an AC-function symbol and let $g$ be a standard function symbol. Let $P = \{X \approx^? a, g(X) \approx^? h(Y), f(Y, W, h(Z)) \approx^? f(c, W)\}$. In this case $V_{>1}(P) = \{Y\}$.

We define proper subterms in order to define admissible subterms in Definition 21.

▶ **Definition 20** (Proper Subterms). *If $t$ is not a pair, we define the proper subterms of $t$, denoted as $PSubterms(t)$ as: $PSubterms(t) = \{s \mid s \in Subterms(t) \text{ and } s \neq t\}$. We define the proper subterm of a pair $\langle t_1, t_2\rangle$ as:*

$$PSubterms(\langle t_1, t_2\rangle) = PSubterms(t_1) \cup PSubterms(t_2).$$

▶ **Definition 21** (Admissible Subterm $AS$). *We say that $s$ is an admissible subterm of a term $t$ if $s$ is a proper subterm of $t$ and $s$ is not a variable. The set of admissible subterms of $t$ is denoted as $AS(t)$. The set of admissible subterms of a unification problem $P$, denoted as $AS(P)$, is defined as $AS(P) = \bigcup_{t \in P} AS(t)$.*

▶ **Example 22.** If $P = \{a \approx^? f(Z_1, Z_2), b \approx^? Z_3, g(h(c), Z) \approx^? Z_4\}$ then $AS(P) = \{h(c), c\}$.

▶ **Definition 23** (Size of a Unification Problem). *We define the size of a term $t$ recursively as follows:*

- $size(a) = 1$
- $size(Y) = 1$
- $size(\langle\rangle) = 1$
- $size(\langle t_1, t_2\rangle) = 1 + size(t_1) + size(t_2)$
- $size(f\ t_1) = 1 + size(t_1)$
- $size(f^{AC}\ t_1) = 1 + size(t_1)$

*Given a unification problem $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$, the size of $P$ is defined as:*

$$size(P) = \sum_{1 \leq i \leq n} size(t_i) + size(s_i).$$

▶ Remark 24 ($s \in AS(t) \implies size(s) < size(t)$). If $s \in AS(t)$, we have that $s$ is a proper subterm of $t$ and therefore the size of $s$ is less than the size of $t$.

**Table 2** Decrease of the components of the lexicographic measure.

| Recursive Call | $\lvert V_{NAC}(P)\rvert$ | $\lvert V_{>1}(P)\rvert$ | $\lvert AS(P)\rvert$ | $size(P)$ |
|---|---|---|---|---|
| line 6, 14 | $<$ | | | |
| lines 9, 15, 18, 23 | $\leq$ | $\leq$ | $\leq$ | $<$ |
| case 1 - line 29 | $\leq$ | $<$ | | |
| case 2 - line 29 | $\leq$ | $\leq$ | $<$ | |
| case 3 - line 29 | $\leq$ | $\leq$ | $\leq$ | $<$ |

## 4.3 Proof Sketch for Termination

### 4.3.1 Non AC Cases

To prove termination of syntactic unification, we can use a lexicographic measure $lex_s$ consisting of two components: $lex_s = (|Vars(P)|, size(P))$, where $Vars(P)$ is the set of variables in the unification problem. We adapted this idea to our proof of termination, by using $|V_{NAC}(P)|$ as our first component and $size(P)$ as the fourth. The proof of termination for all the cases of Algorithm 1 except AC (line 29) is similar to the proof of termination of syntactic unification, with two caveats.

First, we need to use $|V_{NAC}(P)|$ instead of $|Vars(P)|$ to avoid taking into account the variables that are arguments of the AC-function terms introduced by SOLVEAC (see Section 3.2.2). We would still have to take into account the variable terms introduced by SOLVEAC, but those are instantiated by function INSTANTIATESTEP and therefore eliminated from the problem.

Second, in some of the recursive calls (lines 9, 15, 18, 23) we must ensure that the components introduced to prove termination in the AC-case ($|V_{>1}(P)|$ and $|AS(P)|$) do not increase. This is straightforward.

### 4.3.2 The AC-case

Our proof of termination for the AC-case uses the components $|V_{>1}(P)|$ and $|AS(P)|$, proposed in [13]. To explain the choice for the components of the lexicographic measure, let us start by considering the restricted case where $P = \{t \approx^? s\}$. The idea of the proof of termination is to define the set of admissible subterms of a unification problem $AS(P)$ in a way that when we call function SOLVEAC to terms $t$ and $s$, every problem $P_1$ generated will satisfy $|AS(P_1)| < |AS(P)|$.

Let $t_1, \ldots, t_m$ be the arguments of $t$ and let $s_1, \ldots, s_n$ be the arguments of $s$. Then, as described in Section 3.2.2, the left-hand side of $P_1$ is $\{t_1, \ldots, t_m, s_1, \ldots, s_n\}$. Denote by $\{t'_1, \ldots, t'_m, s'_1, \ldots, s'_n\}$ the right-hand side of $P_1$, which means that $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$. This is what motivated our definition of admissible subterms: every term $t'_i$ of the right-hand side of $P_1$ will have $AS(t'_i) = \emptyset$. Therefore, $AS(P_1) \subseteq AS(P)$ always holds.

If we are also in a situation where at least one of the terms in the left-hand side of $P_1$ is not a variable, we can prove that $|AS(P_1)| < |AS(P)|$. To see that, let $u$ be the non-variable term in the left-hand side of $P_1$ of greatest size (if there is a tie, pick any term with greatest size). Then, $u$ is an argument of either $t$ or $s$ and therefore $u \in AS(P)$. We also have $u \notin AS(P_1)$: otherwise there would be a term $u'$ in $P_1$ such that $u \in AS(u')$, which would mean that the size of $u'$ is greater than $u$ (see Remark 24), contradicting our hypothesis that no term in $P_1$ has size greater than $u$. Combining the fact that $AS(P_1) \subseteq AS(P)$ and the fact that there is a term $u$ with $u \in AS(P)$ and $u \notin AS(P_1)$ we obtain that $|AS(P_1)| < |AS(P)|$.

▶ **Example 25.** In the example of Section 2.2, $P = \{f(X, X, Y, a) \approx^? f(b, b, Z)\}$ and we had $AS(P) = \{a, b\}$. After applying SOLVEAC, one of the unification problems that is generated is: $P_1 = \{X \approx^? Z_6, Y \approx^? f(Z_5, Z_5), a \approx^? Z_1, b \approx^? Z_5, Z \approx^? f(Z_1, Z_6, Z_6)\}$, where $AS(P_1) = \emptyset$.

What happens if all the arguments of $t$ and $s$ are variables? In this case we would have $AS(P_1) = AS(P) = \emptyset$ but this is not a problem, since after function SOLVEAC is called, the function INSTANTIATESTEP would execute (receiving as input $P_1$) and it would instantiate all the arguments. The result, call it $P_2$ would be an empty list and we would have $AS(P_2) = AS(P) = \emptyset$ and $size(P_2) < size(P)$.

Therefore, all that is left in this simplified example with only one equation $t \approx^? s$ in the unification problem $P$ is to make sure that when we call INSTANTIATESTEP in a unification problem $P_1$ and obtain as output a unification problem $P_2$ we maintain $|AS(P_2)| \leq |AS(P_1)|$. However, this does not necessarily happen, as Example 26 shows.

▶ **Example 26** (A case where INSTANTIATESTEP increases $|AS|$)**.** Let $f$ and $g$ be AC-function symbols and $P_1 = \{X \approx^? f(Z_1, Z_2), g(X, W) \approx^? g(a, c)\}$. Calling INSTANTIATESTEP with input $P_1$ we obtain $P_2 = \{g(f(Z_1, Z_2), W) \approx^? g(a, c)\}$. In this case we have $AS(P_1) = \{a, c\}$ while $AS(P_2) = \{f(Z_1, Z_2), a, c\}$ and therefore $|AS(P_2)| > |AS(P_1)|$.

This problem motivated the inclusion of the measure $|V_{>1}(P)|$ in our lexicographic measure as we now explain. First, notice that if we changed Example 26 to make it so that $X$ only appears as argument of AC-functions headed by $f$, then instantiating $X$ to an AC-function headed by $f$ would not increase the cardinality of the set of admissible subterms. This is illustrated in Example 27.

▶ **Example 27** (A case where INSTANTIATESTEP does not increase $|AS|$)**.** If we change slightly the problem from Example 26 to $P'_1 = \{X \approx^? f(Z_1, Z_2), f(X, W) \approx^? g(a, c)\}$ and apply INSTANTIATESTEP we would obtain: $P'_2 = \{f(Z_1, Z_2, W) \approx^? g(a, c)\}$, and we would have $AS(P'_1) = AS(P'_2) = \{a, c\}$.

Now, let's go back to our original example of $P = \{t \approx^? s\}$ and $P_1 = \{t_1 \approx^? t'_1, \ldots, t_m \approx^? t'_m, s_1 \approx^? s'_1, \ldots, s_n \approx^? s'_n\}$, and denote by $P_2$ the unification problem obtained by calling INSTANTIATESTEP passing as input $P_1$. We will show that in the cases where $|AS(P_2)|$ may be greater than $|AS(P)|$ we necessarily have $|V_{>1}(P)| > |V_{>1}(P_2)|$.

Consider an arbitrary variable term $X$ on the left-hand side of $P_1$. If $X$ was instantiated by INSTANTIATESTEP, it would be instantiated to an AC-function headed by $f$ (see Section 3.2.2) and therefore would only contribute in increasing $|AS(P_2)|$ in relation with $|AS(P_1)|$ if it also occurred as an argument to a function term (let's call it $t^*$) headed by a different symbol than $f$ (let's say $g$). Since $X$ is in the left-hand side of $P_1$ this means that it was an argument of $t$ or $s$ in $P$ (suppose $t$, without loss of generality) and remember that both $t$ and $s$ are headed by the same symbol $f$. Then $X$ is an argument of $t^*$ and $t$ and therefore, by definition, $X \in V_{>1}(P)$. However $X$ was instantiated by INSTANTIATESTEP and therefore it is not in $V_{>1}(P_2)$. The new variables introduced by SOLVEAC will not make any difference in favour of $|V_{>1}(P_2)|$: when they occur as arguments of function terms, the terms are always headed by the same symbol $f$. Therefore $|V_{>1}(P)| > |V_{>1}(P_2)|$. Accordingly, to fix our problem we include the measure $|V_{>1}(P)|$ before $|AS(P)|$, obtaining the lexicographic measure described in Section 4.2.

The situation described is similar when our unification problem $P$ has more than one equation. Let's say $P = \{t_1 \approx^? s_1, \ldots, t_n \approx^? s_n\}$. The only difference is that it is not enough to call function SOLVEAC and then function INSTANTIATESTEP in only the first equation $t_1 \approx^? s_1$: we need to call function APPLYACSTEP and simplify every equation $t_i \approx^? s_i$.

To see how things may go wrong, notice that in our previous explanation, when the unification problem $P$ had just one equation, a call to SOLVEAC might reduce the admissible subterms by removing a given term (we called it $u$). However, now that $P$ has more than one equation, if $u$ is also present in other equations of the original problem $P$, calling SOLVEAC only in the first equation no longer removes $u$ from the set of admissible subterms.

## 4.4 Soundness and Completeness

As mentioned, to unify terms $t$ and $s$ we use Algorithm 1 with $P = cons((t,s), nil)$, $\sigma = \text{NIL}$ and $V = Vars((t,s))$. However, since the parameters of ACUNIF may change in between the recursive calls, we cannot prove soundness (Corollary 30) directly by induction. We must prove the more general Theorem 29, with generic parameters for the unification problem $P$, the substitution $\sigma$ and the set $V$ of variables that are/were in use. To aid us in this proof we notice that while the recursive calls of ACUNIF may change $P$, $\sigma$ and $V$, some nice relations between them are preserved. These relations between the three components of the input are captured by Definition 28.

▶ **Definition 28** (Nice input). *Given an input $(P, \sigma, V)$, we say that this input is nice if:*

- *$\sigma$ is idempotent*
- *$dom(\sigma) \subseteq V$*
- *$Vars(P) \cap dom(\sigma) = \emptyset$*
- *$Vars(P) \subseteq V$*

▶ **Theorem 29** (Soundness for nice inputs). *Let $(P, \sigma, V)$ be a nice input, and $\delta \in$ ACUNIF$(P, \sigma, V)$. Then, $\delta$ unifies $P$.*

▶ **Corollary 30** (Soundness of ACUNIF). *If $\delta \in$ ACUNIF$(cons((t,s), \text{NIL}), \text{NIL}, Vars((t,s)))$ then $\delta$ unifies $t \approx^? s$.*

Proving completeness of Algorithm 1 boils down to proving Corollary 32 and similarly to the soundness case, this is proved immediately once we prove Theorem 31.

▶ **Theorem 31** (Completeness for nice inputs). *Let $(P, \sigma, V)$ be a nice input, $\delta$ unifies $P$, $\sigma \leq \delta$, and $\delta \subseteq V$. Then, there is a substitution $\gamma \in$ ACUNIF$(P, \sigma, V)$ such that $\gamma \leq_V \delta$.*

▶ **Corollary 32** (Completeness of ACUNIF). *Let $V$ be a set of variables such that $\delta \subseteq V$ and $Vars((t,s)) \subseteq V$. If $\delta$ unifies $t \approx^? s$, then ACUNIF computes a substitution more general than $\delta$, i.e., there is a substitution $\gamma \in$ ACUNIF$(cons((t,s), nil), nil, V)$ such that $\gamma \leq_V \delta$.*

In the proof of completeness, the hypothesis $\delta \subseteq V$ is simply a technicality that was put only in order to guarantee that the new variables introduced by the algorithm do not clash with the variables in $dom(\delta)$ or in the terms in $im(\delta)$ and could be replaced by a different mechanism that guarantees that the variables introduced by the AC-part of ACUNIF are indeed new. As an example, let's go back to the substitutions (see Equation 3) computed in the example of Section 2.2 and notice that the set of variables in the original problem is $V = \{X, Y, Z\}$. If $\delta = \{X \mapsto f(Z_2, a, b), Z \to f(a, Y, Z_2, a, Z_2, a), Z_4 \to c\}$ there is some overlap between the variables in $dom(\delta)$ and in the terms in $im(\delta)$ and the ones introduced by the algorithm, but the substitution $\sigma_4 = \{X \to f(Z_6, b), Z \to f(a, Y, Z_6, Z_6)\}$ that we computed is still more general than $\delta$ (restricted to the variables in $V$). Indeed, if we take $\delta_1 = \{Z_6 \to f(Z_2, a)\}$ then $\delta W = \delta_1 \sigma_4 W$ for all variables $W \in V$.

▶ Remark 33 (High-level description of how to remove hypothesis $\delta \subseteq V$). The key step to prove a variant of Corollary 32 with $V = Vars(t,s)$ and without the hypothesis $\delta \subseteq V$ is to prove that the substitutions computed when we call ACUNIF with input $(P, \sigma, V)$ "differ only by a renaming" from the substitutions computed when we call ACUNIF with input $(P, \sigma, V')$, where $\delta \subseteq V'$. This cannot be proven by induction directly because if $V$ and $V'$ differ and ACUNIF enters the AC-part, the new variables introduced for each input may "differ only by a renaming", i.e. the first component of the two inputs, will also "differ only by a renaming". Once ACUNIF instantiates variables, it may happen that the substitutions

computed so far, i.e. the second component of the two inputs, will also "differ only by a renaming". The solution is to prove by induction the more general statement that if the inputs $(P, \sigma, V)$ and $(P', \sigma', V')$ "differ only by a renaming" then the substitutions computed when we call ACUnif with $(P, \sigma, V)$ "differ only by a renaming" from the substitutions computed when we call ACUnif with $(P', \sigma', V')$.

## 4.5 More Information About the PVS Formalisation

The functions coded in PVS and the statement of the theorems can be found in files `.pvs`, while the proofs of the theorems can be found in the `.prf` files. The PVS theory `unification_alg` contains function ACUnif and the theorems of soundness and completeness; `termination_alg` has the definitions and lemmas needed to prove termination; `apply_ac_step` contains function applyACStep and lemmas about its properties; `aux_unification` contains auxiliary functions such as solveAC and instantiateStep and lemmas about their properties. The PVS theories `diophantine`, `unification`, `substitution`, `equality` and `terms` contain, respectively, definitions and properties about solving linear Diophantine equations, unification, substitutions, equality modulo AC and terms. Finally `list` is a set of parametric theories that define generic functions that operate on lists, not strictly connected to unification.

When specifying functions and theorems, PVS may generate proof obligations to be discharged by the user. These proof obligations are called Type Correctness Conditions (TCCs) and the PVS system includes several pre-defined proof strategies that automatically discharge most of the TCCs. In our code, most TCCs were related to the termination of functions and PVS was able to prove almost all of them automatically. The number of theorems and TCCs proved for each theory, along with the approximate size of each theory and their percentage of the total size is shown in Table 3.

**Table 3** Main Information on the Theories of Our Formalisation.

| Theory | Theorems | TCCs | Size (.pvs) | Size (.prf) | Size (%) |
|---|---|---|---|---|---|
| `unification_alg` | 9 | 18 | 5KB | 1.4MB | 4% |
| `termination_alg` | 80 | 35 | 21KB | 11.0MB | 30% |
| `apply_ac_step` | 23 | 12 | 13KB | 9.0MB | 25% |
| `aux_unification` | 179 | 54 | 52KB | 7.2MB | 20% |
| `Diophantine` | 73 | 44 | 23KB | 1.1MB | 3% |
| `unification` | 75 | 14 | 19KB | 0.8MB | 2% |
| `substitution` | 108 | 16 | 19KB | 1.7MB | 5% |
| `equality` | 67 | 18 | 12KB | 1.1MB | 2% |
| `terms` | 129 | 47 | 27KB | 0.9MB | 2% |
| `list` | 251 | 109 | 52KB | 2.5MB | 6% |
| **Total** | 994 | 367 | 243KB | 36.7MB | 100% |

## 5 Conclusions and Future Work

We have specified Stickel's algorithm [21, 22] for AC-unification in the proof assistant PVS and proved it terminating, sound and complete. Our proof of termination was based on the work of Fages [12, 13]. Since mutual recursion is not straightforward in PVS, we adapted the algorithm to solve an AC-unification problem $P$, instead of only two terms $t$ and $s$.

This introduces some complications in the proof of termination, which we addressed in Section 4.3.2. We have discussed the most interesting points of our formalisation, such as the motivation for the lexicographic measure needed to prove termination.

We envision three possible paths of future work. First, we could extend this first-order algorithm to the nominal setting. A nominal AC-unification algorithm could be used in a logic programming language that employs the nominal setting such as $\alpha$-Prolog [9] or in nominal rewriting [14] and narrowing [4] modulo AC. A second possible path is to use this formalisation as a basis to formalise more efficient algorithms, as discussed in the introduction and in Section 3.2.1. Finally, although PVS does not support code extraction to a programming language such as Haskell or Ocaml, it has the PVSIO feature, which lets us execute a verified algorithm inside the PVS environment and provides input and output operators. Therefore, another possible path is using PVSIO to test existing (or to be developed) implementations of AC-unification.

## References


1 Mohamed Adi and Claude Kirchner. AC-Unification Race: The System Solving Approach, Implementation and Benchmarks. *J. of Sym. Computation*, 14(1):51–70, 1992. doi:10.1016/0747-7171(92)90025-Y.

2 Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Daniele Nantes-Sobrinho, and Ana Cristina Rocha Oliveira. A Formalisation of Nominal $\alpha$-Equivalence with A, C, and AC Function Symbols. *Theor. Comput. Sci.*, 781:3–23, 2019. doi:10.1016/j.tcs.2019.02.020.

3 Mauricio Ayala-Rincón, Washington de Carvalho Segundo, Maribel Fernández, Gabriel Ferreira Silva, and Daniele Nantes-Sobrinho. Formalising Nominal C-Unification Generalised with Protected Variables. *Math. Struct. Comput. Sci.*, 31(3):286–311, 2021. doi:10.1017/S0960129521000050.

4 Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal Narrowing. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016*, page 11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.FSCD.2016.11.

5 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

6 Dan Benanav, Deepak Kapur, and Paliath Narendran. Complexity of Matching Problems. *J. of Sym. Computation*, 3(1/2):203–216, 1987. doi:10.1007/3-540-15976-2_22.

7 Alexandre Boudet. Competing for the AC-Unification Race. *J. of Autom. Reasoning*, 11(2):185–212, 1993. doi:10.1007/BF00881905.

8 Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A New AC Unification Algorithm with an Algorithm for Solving Systems of Diophantine Equations. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 289–299. IEEE Computer Society, 1990. doi:10.1109/LICS.1990.113755.

9 James Cheney and Christian Urban. alpha-Prolog: A Logic Programming Language with Names, Binding and $\alpha$-Equivalence. In *Logic Programming, 20th International Conference, ICLP 2004*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004. doi:10.1007/978-3-540-27775-0_19.

10 Michael Clausen and Albrecht Fortenbacher. Efficient Solution of Linear Diophantine Equations. *J. of Sym. Computation*, 8(1-2):201–216, 1989. doi:10.1016/S0747-7171(89)80025-2.

11 Evelyne Contejean. A Certified AC Matching Algorithm. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *LNCS*, pages 70–84. Springer, 2004. doi:10.1007/978-3-540-25979-4_5.

12 François Fages. Associative-Commutative Unification. In *7th International Conference on Automated Deduction, Napa*, volume 170 of *LNCS*, pages 194–208. Springer, 1984. doi:10.1007/978-0-387-34768-4_12.

13 François Fages. Associative-Commutative Unification. *J. of Sym. Computation*, 3(3):257–275, 1987. doi:10.1016/S0747-7171(87)80004-4.

14 M. Fernández and M. J. Gabbay. *Nominal Rewriting. Information and Computation*, 205(6):917–965, 2007. doi:10.1016/j.ic.2006.12.002.

15 Jean-Pierre Jouannaud and Claude Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321. The MIT Press, 1991.

16 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)*, pages 11–21. IEEE Computer Society, 1992. doi:10.1109/LICS.1992.185515.

17 Florian Meßner, Julian Parsert, Jonas Schöpf, and Christian Sternagel. A Formally Verified Solver for Homogeneous Linear Diophantine Equations. In *Interactive Theorem Proving - 9th International Conference, ITP 2018*, volume 10895 of *LNCS*, pages 441–458. Springer, 2018. doi:10.1007/978-3-319-94821-8_26.

18 Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, volume 607 of *LNCS*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

19 Sam Owre, Natarajan Shankar, John Rushby, and David Stringer-Calvert. PVS Language Reference. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2000. URL: https://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

20 Andrew M Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.

21 Mark E. Stickel. A Complete Unification Algorithm for Associative-Commutative Functions. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, pages 71–76, 1975. URL: http://ijcai.org/Proceedings/75/Papers/011.pdf.

22 Mark E. Stickel. A Unification Algorithm for Associative-Commutative Functions. *J. of the ACM*, 28(3):423–434, 1981. doi:10.1145/322261.322262.


# A Pseudocode for instantiateStep and applyACStep

## A.1 Pseudocode for instantiateStep

Algorithm 2 is the pseudocode for INSTANTIATESTEP. It receives as input a unification problem $P_1$ (the part of our unification problem which we have not yet inspected), a unification problem $P_2$ (the part of our unification problem we have already inspected) and $\sigma$, the substitution computed so far. Therefore, the first call to this function in order to instantiate the unification problem $P$ is with $P_1 = P$, $P_2 = nil$ and $\sigma = nil$. The algorithm returns a triple, where the first component is the remaining unification problem; the second component is the substitution computed by this step; and the third component is a Boolean to indicate if we found an equation $t \approx^? s$ which is not unifiable (in this case the Boolean is $True$) or not (in this case the Boolean is $False$). The only kind of equations that INSTANTIATESTEP identifies as not unifiable are those where one of the terms is a variable, and the other term is a non-variable term that contains this variable. The algorithm works by progressively inspecting every equation $s \approx^? t \in P_1$ and deciding whether:

- One of the terms is a variable and we can instantiate (lines 5-10).
- Both terms are the same variable and we can eliminate this equation from the problem (lines 11-12).

**Algorithm 2** Algorithm that instantiates when possible.

```
1: procedure INSTANTIATESTEP(P1, P2, σ)
2:   if nil?(P1) then return (P2, σ, False)
3:   else
4:     let (t, s) = car(P1), P1' = cdr(P1) in
5:     if (s matches X) and (X not in t) then
6:       σ1 = {X → t}
7:       return INSTANTIATESTEP(σ1 P1', σ1 P2, APPEND(σ1, σ))
8:     else if (t matches X) and (X not in s) then
9:       σ1 = {X → s}
10:      return INSTANTIATESTEP(σ1 P1', σ1 P2, APPEND(σ1, σ))
11:    else if (t matches X) and (X matches s) then
12:      return INSTANTIATESTEP(P1', P2, σ)
13:    else if ((t matches X) and (X in s)) or ((s matches X) and (X in t)) then
14:      return (nil, σ, True)  ▷ the terms t and s are impossible to unify
15:    else
16:      return INSTANTIATESTEP(P1', cons((t, s), P2), σ)  ▷ we skip the equation
```

- The terms are impossible to unify (lines 13-14).
- Neither term is a variable, and so we do not act on this equation (lines 15-16).

## A.2 Pseudocode for applyACStep

▶ Remark 34. In function APPLYACSTEP, we eliminate equations $u \approx^? v$ from our unification problem if $u \approx v$ (line 4). This was done because if we called function SOLVEAC in line 10 of Algorithm 3 passing as parameter two equal terms (modulo AC), the value returned would be $PLst =$ NIL. APPLYACSTEP would interpret that as meaning that the unification pair had no solution (when actually every substitution $\sigma$ is a solution to $\{u \approx^? v\}$) and also return NIL. To prevent this corner case, we eliminate those trivial equations from our unification problem before calling SOLVEAC. In our code, the function EQUAL? tests equality (modulo AC) between terms $t$ and $s$, returning $True$ if the terms are equal and $False$ otherwise.

The first thing APPLYACSTEP does is check if $P_1$ is the null list. If it is (line 2), we have finished applying functions SOLVEAC and INSTANTIATESTEP and we return a list with only one element: $(P_2, \sigma, V)$.

If $P_1$ is not the null list, we get the AC-unification pair in the head of the list (let us call it $(t, s)$) and examine if $t \approx s$. If that is the case (line 4), we simply remove this equation, calling APPLYACSTEP with $(cdr(P_1), P_2, \sigma, V)$.

If $t$ is not equal (modulo AC) to $s$, we call function SOLVEAC. This function will return a list of unification problems $PLst$ (line 7). Next we apply the function INSTANTIATESTEP to every problem $P$ in $PLst$, obtaining a list $ACInstLst$ (lines 8-9), where each entry is a pair $(P', \delta)$. $P'$ is the unification problem after we instantiate the variables and $\delta$ is the substitution computed by this function. It may happen that INSTANTIATESTEP "discovers" that a unification problem is actually unsolvable (this is communicated to APPLYACSTEP via the Boolean value that is part of the output of INSTANTIATESTEP) and in this case this problem is not included in $ACInstLst$.

We check if $ACInstLst$ is null (in this case there are no solutions to the first AC-unification pair, and therefore there are no solutions to the problem) and return NIL if it is. If $ACInstLst$ is not null (lines 12-16), there will be branches to explore. Given

**Algorithm 3** Algorithm for $\textsc{applyACStep}$.

```
1: procedure applyACStep(P_1, P_2, σ, V)
2:     if nil?(P_1) then return cons((P_2, σ, V), NIL)
3:     else let (t, s) = car(P_1) in
4:         if t ≈ s then return applyACStep (cdr(P_1), P_2, σ, V)
5:         else
6:             ▷ assuming t and s are headed by the same function symbol f
7:             PLst = solveAC(t, s, f, V)
8:             ▷ Call instantiateStep in every P in PLst obtaining a list ACInstLst,
9:             ▷ where each entry in this list is a pair (P′, δ).
10:            if nil?(ACInstLst) then return NIL
11:            else
12:                ▷ make an input list InputLst of all the branches we need to explore.
13:                ▷ For each (P′, δ) in ACInstLst, the quadruple in InputLst will be
14:                ▷ (δcdr(P_1), append(P′, δP_2), append(δ, σ), V′) to applyACStep
15:                ▷ recursively explore all the branches
16:                return Flatten(map(applyACStep, InputLst))
```

an entry $(P', \delta)$ of $ACInstLst$, the part of the unification problem to which we must call functions $\textsc{solveAC}$ and $\textsc{instantiateStep}$ is now $\delta cdr(P_1)$ and the part of the unification problem we have already explored is $\textsc{append}(P', \delta P_2)$. The substitution computed so far is $\textsc{append}(\delta, \sigma)$. We take care to update the set of variables that are/were in the problem to include the new variables introduced by $\textsc{solveAC}$ (in Algorithm 3 we change $V$ to $V'$). In short, we make an input list $InputLst$ of all the branches we need to explore and each entry $(P', \delta)$ of $ACInstLst$ gives rise to an entry $(\delta cdr(P_1), \textsc{append}(P', \delta P_2), \textsc{append}(\delta, \sigma), V')$ in $InputLst$.

Finally, $\textsc{applyACStep}$ calls itself recursively taking as argument every input in $InputLst$. This is done by calling $\textsc{map}(\textsc{applyACStep}, InputLst)$ and the output is flattened using function $\textsc{flatten}$.

# B PVS Dependency File Diagram

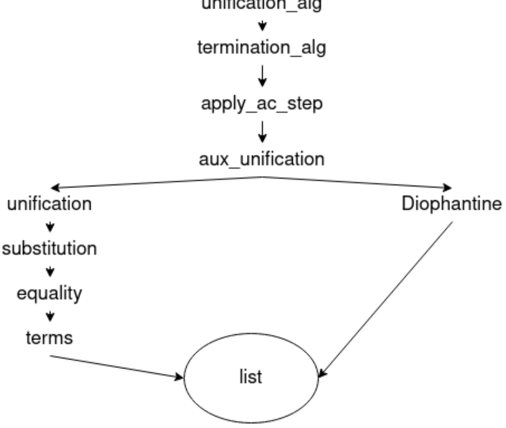



**Figure 1** Dependency Diagram for PVS Theories.

Figure 1 shows the dependency diagram for the PVS theories that compose our formalisation. An arrow going from `theoryA` to `theoryB` means that `theoryA` imports definitions and lemmas from `theoryB`.

▶ Remark 35. The theory `terms` has its definitions and lemmas in the file `terms.pvs` and the proofs of the lemmas in the file `terms.prf`. The same happens for all the theories mentioned in this diagram, except `list`. In our diagram, `list` represents a set of parametric theories that define generic functions (not strictly connected to unification) that operate on lists. The theories in `list` are `list_nat_theory`, `list_theory`, `list_theory2`, `map_theory` and `more_list_theory_props`. However, since the specifics of each theory in `list` is not significant to our formalisation, we grouped them together in our diagram.

▶ Remark 36 (Adapting the Formalisation to More Efficient Algorithms). The dependency diagram of Figure 1 hints on why adapting our formalisation to prove correctness of algorithms that represents terms as DAGs should give us more work than solving the linear Diophantine equations more efficiently. Changing the representation of terms would impact mostly `terms.pvs` but would also require modification in lemmas from other files that are proved by induction on terms. In practice, this means changes in files that depend on `terms.pvs`, specially the ones that more closely depend on `terms.pvs`, such as `equality.pvs`, `substitution.pvs` and `unification.pvs`. In contrast, solving the linear Diophantine equations more efficiently should effectively only require changes in `Diophantine.pvs`. Both adaptations should be faster than starting from scratch.

To further illustrate the additional work of changing the term representation in comparison to solving the linear Diophantine equations more efficiently, let's consider the proof of termination of ACUnif, described in Section 4.2, which is effectively done in file `termination_alg.pvs` (one of the hardest parts of our formalisation, see Table 3). Recalling that the lexicographic measure used is:

$$lex = (|V_{NAC}(P)|,\ |V_{>1}(P)|,\ |AS(P)|,\ size(P))$$

we see that the procedure used to solve the linear diophantine equations plays no role in this proof. In contrast to that, $V_{NAC}(P)$, $V_{>1}(P)$, $AS(P)$, $size(P)$ depend respectively on $V_{NAC}(t)$, $Subterms(t)$ and $size(t)$ which were all defined inductively on the structure of terms and would need to be adjusted in case we changed the way we represent terms.

# An Analysis of Tennenbaum's Theorem in Constructive Type Theory

**Marc Hermes**
Department of Mathematics, Universität des Saarlandes, Saarbrücken, Germany

**Dominik Kirst**
Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

**Abstract**

Tennenbaum's theorem states that the only countable model of Peano arithmetic (PA) with computable arithmetical operations is the standard model of natural numbers. In this paper, we use constructive type theory as a framework to revisit and generalize this result.

The chosen framework allows for a synthetic approach to computability theory, by exploiting the fact that, externally, all functions definable in constructive type theory can be shown computable. We internalize this fact by assuming a version of Church's thesis expressing that any function on natural numbers is representable by a formula in PA. This assumption allows for a conveniently abstract setup to carry out rigorous computability arguments and feasible mechanization.

Concretely, we constructivize several classical proofs and present one inherently constructive rendering of Tennenbaum's theorem, all following arguments from the literature. Concerning the classical proofs in particular, the constructive setting allows us to highlight differences in their assumptions and conclusions which are not visible classically. All versions are accompanied by a unified mechanization in the Coq proof assistant.

**2012 ACM Subject Classification** Theory of computation → Constructive mathematics

**Keywords and phrases** first-order logic, Peano arithmetic, Tennenbaum's theorem, constructive type theory, Church's thesis, synthetic computability, Coq

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.9

**Supplementary Material** *Software*: `https://www.ps.uni-saarland.de/extras/tennenbaum`

## 1 Introduction

In classical logic, it is relatively straightforward to establish the existence of non-standard models of first-order Peano arithmetic (PA), showing that the theory does not possess a unique model up to isomorphism and is therefore not categorical. Following a typical textbook presentation [3], one way to construct a non-standard model is by adding a new constant symbol $c$ to the language of PA together with the enumerable list of new axioms $c \neq 0$, $c \neq 1$, $c \neq 2$, etc. This yields a theory with the property that every finite subset of its axioms is satisfied by the standard model $\mathbb{N}$, since we can always give a large enough interpretation of the constant $c$ in $\mathbb{N}$. Hence by the compactness theorem, the full theory has a model $\mathcal{M}$, which must then be non-standard, as the interpretation of $c$ in $\mathcal{M}$ corresponds to an element which is larger then any number $n : \mathbb{N}$.

This construction comes with some striking consequences. Since PA can prove that for every bound $n$, the products of the form $\prod_{k \leq n} a_k$ exist, the presence of the non-standard element $c$ in $\mathcal{M}$ gives rise to infinite products $\prod_{k \leq c} a_k$. The general PA model $\mathcal{M}$ can therefore exhibit behaviors disagreeing with the usual intuition that computations in PA are finitary, which are largely based on the familiarity with the standard model $\mathbb{N}$.

However, these intuitions are not too far off the mark, as was demonstrated by Stanley Tennenbaum [38] in a remarkable theorem: $\mathbb{N}$ is (up to isomorphism) the only *computable* model of first-order PA. Here, a model is considered *computable* if its elements can be

© Marc Hermes and Dominik Kirst;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 9; pp. 9:1–9:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

coded by numbers in $\mathbb{N}$, and the arithmetic operations on model elements can be realized by computable functions on these codes. Usually, this theorem is formulated in a classical framework such as ZF set theory and the precise meaning of *computable* is given by making reference to a concrete model of computation like Turing machines, $\mu$-recursive functions, or the $\lambda$-calculus [14, 34]. But as is custom, the computability of a function is rarely proven by exhibiting an explicit construction in the chosen model, but by a call to the *Church-Turing thesis*, expressing that every function intuitively computable will be computable in the model.

To offer an alternative and more rigorous perspective, in this paper we revisit Tennenbaum's theorem in constructive type theory. Since we can externally observe that all functions of constructive type theory are computable, we have the freedom to simply treat every function as being computable, without exhibiting any internal representation in a formal model of computation. This is known as the *synthetic* approach to computability [31, 1] and simplifies computability arguments to the point where the above-mentioned intuitions usually suffice to give complete proofs with no formal gaps, and renders mechanization much more feasible.

This also leads to a simplification as it comes to the statement of Tennenbaum's theorem: In the most natural semantics interpreting the arithmetic operations with type-theoretic functions, simply *all* models are computable and we no longer need "computable model" as part of the theorem statement. We furthermore *internalize* computability by assuming a version of *Church's thesis* [18, 40, 7], an axiom which expresses that *all* functions $\mathbb{N} \to \mathbb{N}$ have a representation in an internally captured formalism, in our case PA. With this setup, all arguments involving a computability proof reduce to the constructions of type-theoretic functions, giving a formal counterpart to the informal appeal to the Church-Turing thesis.

Based on this framework, we follow the classical presentations of Tennenbaum's theorem [14, 34] to develop constructive versions only assuming a type-theoretic version of *Markov's principle* [21]. Classically, these proofs all yield the same version of Tennenbaum's theorem, but under a constructive lens, they differ in the strength of their respective assumptions and conclusions. This is then complemented by the adaption of an inherently constructive variant given by McCarty [23, 24].

Concretely, our contributions can be summarized as follows:

- We formulate, establish, and compare several versions of Tennenbaum's theorem in the setting of synthetic computability based on constructive type theory.
- We generalize Tennenbaum's theorem to models with decidable divisibility relation that need not be computable in general nor even enumerable (Corollary 43).
- We provide a Coq mechanization covering all results studied in this paper.[1]

To make the paper self-contained, we start out in Section 2 by giving a quick introduction to the essential features of constructive type theory, synthetic computability, and the type-theoretic specification of first-order logic. We continue with a presentation of the first-order axiomatization of PA as given in previous work [15], and of basic results about its standard and non-standard models in Section 4. These are then used in Section 5 to establish results that allow the encoding of predicates on $\mathbb{N}$ in non-standard models, which are essential in the proof of Tennenbaum's theorem. In Section 6 we introduce the chosen formulation of Church's thesis, which is then used to derive Tennenbaum's theorem in several variations in Section 7. We conclude in Section 8 with observations about these proofs and remarks on the Coq mechanization as well as related and future work.

---

[1] The only two facts with no formal counterpart in Coq are clearly marked as "Hypothesis" in Section 7.3. The full mechanization is accessible from the web page listed as supplementary material and systematically hyperlinked with the highlighted statements in the PDF version of this paper.

# 2 Preliminaries

## 2.1 Constructive Type Theory

Our framework is the calculus of inductive constructions (CIC) [6, 27] which is implemented in the Coq proof assistant [37], providing a predicative hierarchy of *type universes* above a single impredicative universe $\mathbb{P}$ of *propositions* and the capability of inductive type definitions. On type level, we have the unit type $\mathbb{1}$ with a single element, the void type $\mathbb{0}$, function spaces $X \to Y$, products $X \times Y$, sums $X + Y$, dependent products[2] $\forall(x : X).\, A\, x$, and dependent sums $\Sigma(x : X).\, A\, x$. On the propositional level, the notions as listed in the order above, are denoted by the usual logical notation ($\top$, $\bot$, $\to$, $\land$, $\lor$, $\forall$, $\exists$).[3] It is important to note that the so-called *large eliminations* from the impredicative $\mathbb{P}$ into higher types of the hierarchy are restricted. In particular it is therefore generally not possible to show $(\exists x.\, p\, x) \to \Sigma x.\, p\, x$.[4] The restriction does however allow for large elimination of the equality predicate $= : \forall X.\, X \to X \to \mathbb{P}$, as well as function definitions by well-founded recursion.

We will also use the basic inductive types of *Booleans* ($\mathbb{B} := \mathsf{tt} \mid \mathsf{ff}$), *Peano natural numbers* ($n : \mathbb{N} := 0 \mid n + 1$), the *option type* ($\mathcal{O}(X) := {^\circ}x \mid \emptyset$) and *lists* ($l : \mathsf{List}(X) := [\,] \mid x :: l$). Furthermore, by $X^n$ we denote the type of *vectors* $\vec{v}$ of length $n : \mathbb{N}$ over $X$.

▶ **Definition 1.** *A proposition $P : \mathbb{P}$ is called* definite *if $P \lor \neg P$ holds and* stable *if $\neg\neg P \to P$. The same terminology is used for predicates $p : X \to \mathbb{P}$ given they are pointwise* definite *or* stable*. We furthermore want to recall the following logical principles:*

$$\mathsf{LEM} := \forall P : \mathbb{P}.\, \mathsf{definite}\, P \qquad \textit{(Law of Excluded Middle)}$$
$$\mathsf{DNE} := \forall P : \mathbb{P}.\, \mathsf{stable}\, P \qquad \textit{(Double Negation Elimination)}$$
$$\mathsf{MP} := \forall f : \mathbb{N} \to \mathbb{N}.\, \mathsf{stable}\, (\exists n.\, f n = 0) \qquad \textit{(Markov's Principle)}$$

*all of which are not provable in* CIC*.*

Note that LEM and DNE are equivalent while MP is much weaker and has a constructive interpretation [21]. For convenience, and as used for instance by Bauer [2], we adapt the reading of double negated statements like $\neg\neg P$ as "*potentially* $P$".[5]

▶ Remark (Handling $\neg\neg$). Given any propositions $A, B$ we constructively have $(A \to \neg B) \leftrightarrow (\neg\neg A \to \neg B)$, telling us that whenever we are trying to prove a negated goal, we can remove double negations in front of any available assumption. More specifically then, any statement of the form $\neg\neg A_1 \to \ldots \to \neg\neg A_n \to \neg\neg C$, is equivalent to $A_1 \to \ldots \to A_n \to \neg\neg C$ and since $C \to \neg\neg C$ holds, it furthermore suffices to show $A_1 \to \ldots \to A_n \to C$ in this case. In the following, we will make use of these facts without further notice.

## 2.2 Synthetic Computability

As already expressed in Section 1, constructive type theory allows us to interpret all definable functions as computable. We then get simplified versions [9] of usual definitions:

---

[2] As is custom in Coq, we write $\forall$ in place of the symbol $\Pi$ for dependent products.
[3] Negation $\neg A$ is used as an abbreviation for both $A \to \bot$ and $A \to \mathbb{0}$.
[4] The direction $(\Sigma x.\, p\, x) \to \exists x.\, p\, x$ is however always provable. Intuitively, one can think of $\exists x.\, p\, x$ as stating the mere existence of some value satisfying $p$, while $\Sigma x.\, p\, x$ is a type that also carries a value satisfying this.
[5] $\neg\neg P$ expresses the impossibility of $P$ being wrong, so it represents a guarantee that $P$ can potentially be shown correct.

▶ **Definition 2** (Enumerability). *Let $p : X \to \mathbb{P}$ be some predicate. We say that $p$ is* enumerable *if there is an* enumerator $f : \mathbb{N} \to \mathcal{O}(X)$ *such that* $\forall x{:}X.\, p\, x \leftrightarrow \exists n.\, f n = {}^\circ x$.

▶ **Definition 3** (Decidability). *Let $p : X \to \mathbb{P}$ be some predicate. We call $f : X \to \mathbb{B}$ a* decider *for $p$ and write* $\mathsf{decider}\, p\, f$ *iff* $\forall x{:}X.\, p\, x \leftrightarrow f x = \mathsf{tt}$. *We then define the following notions of decidability:*

- $\mathsf{Dec}\, p := \exists f{:}X \to \mathbb{B}.\, \mathsf{decider}\, p\, f$
- $\mathsf{dec}(P{:}\mathbb{P}) := P + \neg P$.

*In both cases we will often refer to the predicate or proposition simply as being* decidable.

We also expand the synthetic vocabulary with notions for types. In the textbook setting, many of them can only be defined for sets which are in bijection with $\mathbb{N}$, but synthetically they can be handled in a more uniform way.

▶ **Definition 4.** *We call a type $X$*

- enumerable *if $\lambda x{:}X.\top$ is enumerable,*
- discrete *if there exists a decider for equality $=$ on $X$,*
- separated *if there exists a decider for apartness $\neq$ on $X$,*
- witnessing *if* $\forall f{:}X \to \mathbb{B}.\, (\exists x.\, f x = \mathsf{tt}) \to \Sigma x.\, f x = \mathsf{tt}$.

▶ **Fact 5.** *In the particular type theory we use, $\mathbb{N}$ is witnessing.*

## 2.3 First-Order Logic

In order to study Tennenbaum's theorem, we need to give a description of the first-order theory of PA and the associated intuitionistic theory of *Heyting arithmetic* (HA), which has the same axiomatization, but uses intuitionistic first-order logic. We follow prior work in [9, 10, 15] and describe first-order logic inside of the constructive type theory, by inductively defining formulas, terms, and the deduction system. We then define a semantics for this logic, which uses Tarski models and interprets formulas over the respective domain of the model. The type of natural numbers $\mathbb{N}$ will then naturally be a model of HA.

Before specializing to one particular theory, we keep the definition of first-order logic general and fix some arbitrary signature $\Sigma = (\mathcal{F}; \mathcal{P})$ for function and predicate symbols.

▶ **Definition 6** (Terms and Formulas). *We define terms $t{:}\mathsf{tm}$ and formulas $\varphi{:}\mathsf{fm}$ inductively.*

$$s, t{:}\mathsf{tm} ::= x_n \mid f\, \vec{v} \qquad (n{:}\mathbb{N},\ f{:}\mathcal{F},\ \vec{v}{:}\mathsf{tm}^{|f|})$$
$$\alpha, \beta{:}\mathsf{fm} ::= \bot \mid P\, \vec{v} \mid \alpha \to \beta \mid \alpha \wedge \beta \mid \alpha \vee \beta \mid \forall\, \alpha \mid \exists\, \beta \qquad (P{:}\mathcal{P},\ \vec{v}{:}\mathsf{tm}^{|P|}).$$

*Where $|f|$ and $|P|$ are the arities of the function symbol $f$ and predicate symbol $P$, respectively.*

We use de Bruijn indexing to formalize the binding of variables to quantifiers. This means that the variable $x_n$ at some position in a formula is *bound* to the $n$-th quantifier preceding this variable in the syntax tree of the formula. If there is no quantifier binding the variable, it is said to be *free*.

▶ **Definition 7** (Substitution). *Given a variable assignment $\sigma : \mathbb{N} \to \mathsf{tm}$ we recursively define* substitution *on terms by $x_k[\sigma] := \sigma\, k$ and $f\, \vec{v} := f(\vec{v}[\sigma])$, further extended to formulas by*

$$\bot[\sigma]{:=}\bot \qquad (P\, \vec{v})[\sigma]{:=}P\, (\vec{v}[\sigma]) \qquad (\alpha \,\dot{\Box}\, \beta)[\sigma]{:=}\alpha[\sigma] \,\dot{\Box}\, \beta[\sigma] \qquad (\dot{\nabla}\, \varphi)[\sigma]{:=}\dot{\nabla}(\varphi[x_0; \lambda x.(\sigma x)[\uparrow]])$$

*where $\dot{\Box}$ is any logical connective and $\dot{\nabla}$ any quantifier. The expression $x;\sigma$ is defined by $(x;\sigma)\, 0 := x$ as well as $(x;\sigma)(n+1) := \sigma\, n$ and is simply appending $x$ as the first element to $\sigma{:}\mathbb{N} \to \mathsf{tm}$. By $\uparrow$ we designate the substitution $\lambda n.\, x_{n+1}$ shifting all variable indices by one.*

▶ **Definition 8** (Natural Deduction). *Natural deduction* $\vdash : (\mathsf{fm} \to \mathbb{P}) \to \mathsf{fm} \to \mathbb{P}$ *is characterized inductively by the usual rules (see Appendix A). We write* $\vdash$ *for intuitionistic natural deduction and* $\vdash_c$ *for the classical variant, extending* $\vdash$ *with Peirce's law* $((\varphi \to \psi) \to \varphi) \to \varphi$.

▶ **Definition 9** (Tarski Semantics). *A* model $\mathcal{M}$ *consists of a type* $D$ *designating its domain together with functions* $f^{\mathcal{M}} : D^{|f|} \to D$ *and* $P^{\mathcal{M}} : D^{|P|} \to \mathbb{P}$ *for all symbols* $f$ *in* $\mathcal{F}$ *and* $P$ *in* $\mathcal{P}$*. We will also use* $\mathcal{M}$ *to refer to the domain. Functions* $\rho : \mathbb{N} \to \mathcal{M}$ *are called environments and are used as variable assignments to recursively give evaluations to terms:*

$$\hat{\rho}\, x_k := \rho\, k \qquad \hat{\rho}\,(f\,\vec{v}) := f^{\mathcal{M}}(\hat{\rho}\,\vec{v}) \qquad (v : \mathsf{tm}^n)$$

*This interpretation is then extended to formulas via the satisfaction relation:*

$$\begin{aligned} \mathcal{M} \vDash_\rho P\,\vec{v} &:= P^{\mathcal{M}}(\hat{\rho}\,\vec{v}) & \mathcal{M} \vDash_\rho \alpha \to \beta &:= \mathcal{M} \vDash_\rho \alpha \to \mathcal{M} \vDash_\rho \beta \\ \mathcal{M} \vDash_\rho \alpha \land \beta &:= \mathcal{M} \vDash_\rho \alpha \land \mathcal{M} \vDash_\rho \beta & \mathcal{M} \vDash_\rho \alpha \lor \beta &:= \mathcal{M} \vDash_\rho \alpha \lor \mathcal{M} \vDash_\rho \beta \\ \mathcal{M} \vDash_\rho \forall \alpha &:= \forall x : D.\ \mathcal{M} \vDash_{x;\rho} \alpha & \mathcal{M} \vDash_\rho \exists \alpha &:= \exists x : D.\ \mathcal{M} \vDash_{x;\rho} \alpha \end{aligned}$$

*We say that a formula* $\varphi$ holds in the model $\mathcal{M}$ *and write* $\mathcal{M} \vDash \varphi$ *if for every* $\rho$ *we have* $\mathcal{M} \vDash_\rho \varphi$*. We extend this notation to theories* $\mathcal{T} : \mathsf{fm} \to \mathbb{P}$ *by writing* $\mathcal{M} \vDash \mathcal{T}$ *iff* $\forall \varphi.\, \mathcal{T}\,\varphi \to \mathcal{M} \vDash \varphi$ *and we write* $\mathcal{T} \vDash \varphi$ *if* $\mathcal{M} \vDash \varphi$ *for all models* $\mathcal{M}$ *with* $\mathcal{M} \vDash \mathcal{T}$.

▶ **Fact 10** (Soundness). *For any formula* $\varphi$ *and theory* $\mathcal{T}$*, if* $\mathcal{T} \vdash \varphi$ *then* $\mathcal{T} \vDash \varphi$.

From the next section onwards, we will no longer explicitly write formulas with de Bruijn indices, but will use the conventional notation which uses named variables.

## 3 Axiomatization of Peano Arithmetic

We present PA following [15], as a first-order theory with a signature consisting of symbols for the constant zero, the successor function, addition, multiplication and equality:

$$\Sigma_{\mathsf{PA}} := (\mathcal{F}_{\mathsf{PA}}; \mathcal{P}_{\mathsf{PA}}) = (0,\ S, +, \times; =)$$

The finite core of PA axioms consists of statements characterizing the successor function, as well as addition and multiplication:

$$\begin{aligned} \text{Disjointness} &: \forall x.\, Sx = 0 \to \bot & \text{Injectivity} &: \forall xy.\, Sx = Sy \to x = y \\ +\text{-base} &: \forall x.\, 0 + x = x & +\text{-recursion} &: \forall xy.\, (Sx) + y = S(x + y) \\ \times\text{-base} &: \forall x.\, 0 \times x = 0 & \times\text{-recursion} &: \forall xy.\, (Sx) \times y = y + x \times y \end{aligned}$$

We then get the full (and infinite) axiomatization of PA with the axiom scheme of induction for unary formulas. In our meta-theory the schema is a type-theoretic function on formulas:

$$\lambda \varphi.\, \varphi[0] \to (\forall x.\, \varphi[x] \to \varphi[Sx]) \to \forall x.\, \varphi[x]$$

If instead of the induction scheme we add the axiom $\forall x.\, x = 0 \lor \exists y.\, x = Sy$, we get the theory Q known as *Robinson arithmetic*. Both PA and Q also contain axioms for equality:

$$\begin{aligned} \text{Reflexivity} &: \forall x.\, x = x \\ \text{Symmetry} &: \forall xy.\, x = y \to y = x \\ \text{Transitivity} &: \forall xyz.\, x = y \to y = z \to x = z \\ S\text{-equality} &: \forall xy.\, x = y \to Sx = Sy \\ +\text{-equality} &: \forall xyuv.\, x = u \to y = v \to x + y = u + v \\ \times\text{-equality} &: \forall xyuv.\, x = u \to y = v \to x \times y = u \times v \end{aligned}$$

The classical first-order theory of Peano arithmetic is described by $\mathsf{PA} \vdash_c$, while its intuitionistic counterpart – Heyting arithmetic – is given by $\mathsf{PA} \vdash$.[6] Since the constructive type theory we have chosen to work in only gives us a model for Heyting arithmetic, we will only work with the intuitionistic theory $\mathsf{PA} \vdash$. To emphasize this we will from now on write HA instead of PA.

For simplicity, we only consider models that interpret the equality symbol with the actual equality relation of its domain, so-called *extensional* models. Note that in the Coq development we even make the equality symbol a syntactic primitive, therefore enabling the convenient behavior that the interpreted equality reduces to actual equality.

▶ **Definition 11.** *We recursively define a function* $\overline{\cdot} : \mathbb{N} \to \mathsf{tm}$ *by* $\overline{0} := 0$ *and* $\overline{n+1} := S\overline{n}$, *giving every natural number a representation as a term. Any term $t$ which is of the form $\overline{n}$ will be called* numeral*.*

We furthermore use notations for expressing *less than* $x < y := \exists k.\, S(x+k) = y$, *less or equal* $x \leq y := \exists k.\, x + k = y$ and for *divisibility* $x \mid y := \exists k.\, x \times k = y$.

The formulas of HA can be classified in a hierarchy based on the their computational properties. We will only consider two levels of this hierarchy, namely $\Delta_1$ and $\Sigma_1$ formulas:

▶ **Definition 12.** *A formula $\varphi$ is $\Delta_1$ if we have $\mathsf{HA} \vdash \varphi \lor \neg\varphi$ and if for every substitution $\sigma$ such that $\varphi[\sigma]$ is closed we even have $\mathsf{Q} \vdash \varphi[\sigma]$ or $\mathsf{Q} \vdash \neg\varphi[\sigma]$.*

*We call a formula $\exists_n$ if it is of the form $\exists \ldots \exists \varphi_0$, where $\varphi_0$ is a $\Delta_1$ formula preceded by exactly $n$ existential quantifiers. If a formula is $\exists_n$ for some $n$, it is called $\Sigma_1$.*

Note that every $\exists_n$ formula can be proven equivalent to a $\exists_1$ formula by replacing the $n$ quantifiers $\exists x_1 \ldots \exists x_n$ with $\exists x\, \exists x_1 < x \ldots \exists x_n < x$. A more syntactic definition of $\Delta_1$ would characterize them as the formulas which are equivalent to both a $\Pi_0$ and $\Sigma_0$ formula. For our purposes the more semantic definition simply stipulating the necessary decidability properties is preferable, as it directly implies the absoluteness properties we will actually need:

▶ **Lemma 13** ($\Delta_1$-Absoluteness)**.** *Let $\mathcal{M} \vDash \mathsf{HA}$ and $\varphi$ be any closed $\Delta_1$ formula, then we have $\mathbb{N} \vDash \varphi \leftrightarrow \mathcal{M} \vDash \varphi$.*

**Proof.** By Definition 12 we have either $\mathsf{HA} \vdash \varphi$ or $\mathsf{HA} \vdash \neg\varphi$. Since $\mathbb{N} \vDash \varphi$ we must have $\mathsf{HA} \vdash \varphi$ and therefore $\mathcal{M} \vDash \varphi$ by soundness. ◀

▶ **Lemma 14** ($\Sigma_1$-Completeness)**.** *For any unary $\Delta_1$ formula $\varphi(x)$ we have $\mathbb{N} \vDash \exists x.\, \varphi(x)$ iff $\mathsf{HA} \vdash \exists x.\, \varphi(x)$.*

**Proof.** The assumption $\mathbb{N} \vDash \exists x.\, \varphi(x)$ gives us $n : \mathbb{N}$ with $\mathbb{N} \vDash \varphi(\overline{n})$. By Lemma 13 we then have $\mathsf{HA} \vdash \varphi(\overline{n})$, which in turn shows $\mathsf{HA} \vdash \exists x.\, \varphi(x)$. The converse follows by soundness. ◀

## 4 Standard and Non-standard Models of HA

From now on $\mathcal{M}$ will always designate a HA model. We will now see that there is a canonical way to embed $\mathbb{N}$ into any model of PA.

▶ **Fact 15.** *We recursively define a function $\nu : \mathbb{N} \to \mathcal{M}$ by $\nu\, 0 := 0^{\mathcal{M}}$ and $\nu\,(n+1) := S^{\mathcal{M}}(\nu\; n)$. We define the predicate $\mathsf{std} := \lambda e.\, \exists n. \overline{n} = e$ and refer to $e$ as a* standard number *if $\mathsf{std}\, e$ and* non-standard *if $\neg\, \mathsf{std}\, e$. We then have*

[6] Another way to treat the distinction between classical and intuitionistic theories would be to add all instances of Peirce's law to the axioms of a theory, instead of building them into the deduction system.

1. $\hat{\rho}\,\overline{n} = \nu\, n$ *for any* $n\!:\!\mathbb{N}$ *and environment* $\rho\!:\!\mathbb{N} \to \mathcal{M}$.
2. $\nu$ *is an injective homomorphism and therefore an* embedding *of* $\mathbb{N}$ *into* $\mathcal{M}$.

*We take both facts as a justification to abuse notation and also write* $\overline{n}$ *for* $\nu\, n$.

Usually we would have to write $0^{\mathcal{M}}, S^{\mathcal{M}}, +^{\mathcal{M}}, \times^{\mathcal{M}}, =^{\mathcal{M}}$ for the interpretations of the respective symbols in a model $\mathcal{M}$. For better readability we will however take the freedom to overload the symbols $0, S, +, \cdot, =$ to also refer to these interpretations.

▶ **Definition 16.** *$\mathcal{M}$ is called a* standard model *if there is a bijective homomorphism* $\varphi : \mathbb{N} \to \mathcal{M}$. *We will accordingly write* $\mathcal{M} \cong \mathbb{N}$ *if this is the case.*

We can show that $\nu$ is essentially the only homomorphism from $\mathbb{N}$ to $\mathcal{M}$ we need to worry about, since it is unique up to functional extensionality:

▶ **Lemma 17.** *Let* $\varphi : \mathbb{N} \to \mathcal{M}$ *be a homomorphism, then* $\forall x\!:\!\mathbb{N}.\, \varphi\, x = \nu\, x$.

**Proof.** By induction on $x$ and using the fact that both are homomorphisms. ◀

We now have two equivalent ways to express standardness of a model.

▶ **Lemma 18.** $\mathcal{M} \cong \mathbb{N}$ *iff* $\forall e\!:\!\mathcal{M}.\, \mathsf{std}\, e$.

**Proof.** Given $\mathcal{M} \cong \mathbb{N}$, there is an isomorphism $\varphi : \mathbb{N} \to \mathcal{M}$. Since $\varphi$ is surjective, Lemma 17 implies that $\nu$ must also be surjective. For the converse: if $\nu$ is surjective, it is an isomorphism since it is injective by Fact 15. ◀

Having seen that every model contains a unique embedding of $\mathbb{N}$, one may wonder whether there is a formula $\varphi$ which could define and pick out precisely the standard numbers in $\mathcal{M}$. Lemma 19 gives a negative answer to this question:

▶ **Lemma 19.** *There is a unary formula* $\varphi(x)$ *with* $\forall e\!:\!\mathcal{M}.\, \big(\mathsf{std}\, e \leftrightarrow \mathcal{M} \vDash \varphi(e)\big)$ *if and only if* $\mathcal{M} \cong \mathbb{N}$.

**Proof.** Given a formula $\varphi$ with the stated property, we certainly have $\mathcal{M} \vDash \varphi(\overline{0})$ since $\overline{0}$ is a standard number, and clearly $\mathcal{M} \vDash \varphi(x) \implies \mathsf{std}\, x \implies \mathsf{std}\,(Sx) \implies \mathcal{M} \vDash \varphi(Sx)$. Thus by induction in the model, we have $\mathcal{M} \vDash \forall x.\, \varphi(x)$, which is equivalent to $\forall e\!:\!\mathcal{M}.\, \mathsf{std}\, e$. The converse implication holds by choosing the formula $x = x$. ◀

We now turn our attention to models which are not isomorphic to $\mathbb{N}$.

▶ **Fact 20.** *For any* $e\!:\!\mathcal{M}$, *we have* $\neg\, \mathsf{std}\, e$ *iff* $\forall n\!:\!\mathbb{N}.\ e > \overline{n}$.

▶ **Definition 21.** *Founded on the result of Fact 20 we write* $e > \mathbb{N}$ *iff* $\neg\, \mathsf{std}\, e$ *and call* $\mathcal{M}$

- non-standard *(written* $\mathcal{M} > \mathbb{N}$*) iff there is* $e\!:\!\mathcal{M}$ *such that* $e > \mathbb{N}$,
- not standard *(written* $\mathcal{M} \ncong \mathbb{N}$*) iff* $\neg \mathcal{M} \cong \mathbb{N}$.

*We will also write* $e\!:\!\mathcal{M} > \mathbb{N}$ *to express the existence of a non-standard element* $e$ *in* $\mathcal{M}$.

Of course we have $\mathcal{M} > \mathbb{N} \to \mathcal{M} \ncong \mathbb{N}$, but the converse implication does not hold constructively in general, so the distinction of both notions becomes meaningful.

▶ **Lemma 22** (Overspill). *If* $\mathcal{M} \ncong \mathbb{N}$ *and* $\varphi(x)$ *is unary with* $\mathcal{M} \vDash \varphi(\overline{n})$ *for every* $n\!:\!\mathbb{N}$, *then*

1. $\neg\, \big(\forall e\!:\!\mathcal{M}.\, \mathcal{M} \vDash \varphi(e) \to \mathsf{std}\, e\big)$
2. $\mathsf{stable\ std} \to \neg\neg\, \exists\, e > \mathbb{N}.\, \mathcal{M} \vDash \varphi(e)$
3. $\mathsf{DNE} \to \exists\, e > \mathbb{N}.\, \mathcal{M} \vDash \varphi(e)$.

**Proof.** (1) Assuming $\forall e{:}\mathcal{M}.\ \mathcal{M} \vDash \varphi(e) \to \mathsf{std}\, e$ and combining it with our assumption that $\varphi$ holds on all numerals, Lemma 19 implies $\mathcal{M} \cong \mathbb{N}$, giving us a contradiction. For (2) note that we constructively have that $\neg\exists e{:}\mathcal{M}.\ \neg\mathsf{std}\, e \wedge \mathcal{M} \vDash \varphi(e)$ implies $\forall e{:}\mathcal{M}.\ \mathcal{M} \vDash \varphi(e) \to \neg\neg\,\mathsf{std}\, e$, and by using the stability of std we therefore get a contradiction in the same way as in (1). Statement (3) immediately follows from (2). ◀

In Section 5 we will use Overspill to encode arbitrary predicates by non-standard elements.

## 5 Coding Finite and Infinite Predicates

There is a standard way in which finite sets of natural numbers can be encoded by a single natural number. This is readily established in $\mathbb{N}$ and can then be carried over with relative ease to any HA model. Overspill has interesting consequences when it comes to this encoding, as for models $\mathcal{M} \not\cong \mathbb{N}$, it allows the potential encoding of any predicate $p : \mathbb{N} \to \mathbb{P}$.

For the natural number version of the encoding, we only need some injective function $\pi : \mathbb{N} \to \mathbb{N}$ whose image consists only of prime numbers.

▶ **Lemma 23** (Finite Coding in $\mathbb{N}$). *Given any predicate $p : \mathbb{N} \to \mathbb{P}$ and bound $n{:}\mathbb{N}$, we have*

$$\neg\neg\,\exists c{:}\mathbb{N}\ \forall u{:}\mathbb{N}.\big(u < n \to (p\,u \leftrightarrow \pi_u \mid c)\big) \wedge \big(\pi_u \mid c \to u < n\big)$$

*i.e. up to the specified bound $n$, the code $c$ is divisible by the prime $\pi_u$ iff $p$ holds on $u{:}\mathbb{N}$. The second part of the conjunction assures that no primes bigger than $\pi_n$ are present in the code. Note that if $p$ is definite, we can drop the $\neg\neg$.*

**Proof.** We do a proof by induction on $n$. For $n = 0$ we can choose $c = 1$. For the induction step we first note that $\neg\neg(p\,n \vee \neg p\,n)$ is constructively provable and that the induction hypothesis as well as the goal come with double negations at the front. Using $p\,n \vee \neg p\,n$ we can now consider two cases. If $\neg p\,n$ we can simply take the code $c$ given by the induction hypothesis. If $p\,n$, we set the new code to be $c \cdot \pi_n$. In both cases the separate parts of the conjunction are checked by making use of the fact that $\pi$ is an injective prime function. ◀

▶ **Remark 24.** *To formulate the above result in a generic model $\mathcal{M} \vDash \mathsf{HA}$, we require an object level representation of the prime function $\pi$. For now we will simply assume that we have such a binary formula $\Pi(x, y)$ and defer the justification to Section 6.*

This now makes it possible to express "$\pi_u$ divides $c$" by $\exists p.\ \Pi(u, p) \wedge p \mid c$, where we will abuse notation and simply write $\Pi(u) \mid c$ for this. With $\Pi$ then, we can take the coding result established for $\mathbb{N}$ and use it to show a similar result in any model of HA.

▶ **Lemma 25** (Finite Coding in $\mathcal{M}$). *For any binary formula $\alpha(x, y)$ and $n{:}\mathbb{N}$ we have*

$$\mathcal{M} \vDash \forall b\,\neg\neg\,\exists c\,\forall u < \overline{n}.\ \alpha(u, b) \leftrightarrow \Pi(u) \mid c.$$

*If $\mathcal{M} \vDash \alpha(\overline{u}, b)$ is definite for every $u{:}\mathbb{N}$, $b{:}\mathcal{M}$, we can drop the $\neg\neg$ in the above.*

**Proof.** Let $b : \mathcal{M}$, then define the predicate $p := \lambda u : \mathbb{N}.\ \mathcal{M} \vDash \alpha(\overline{u}, b)$. Then Lemma 23 potentially gives us a code $a{:}\mathbb{N}$ for $p$ up to the bound $n$. It now suffices to show that the actual existence of $a{:}\mathbb{N}$ already implies

$$\mathcal{M} \vDash \exists c\,\forall u < \overline{n}.\ \alpha(u, b) \leftrightarrow \Pi(u) \mid c.$$

And indeed, we can verify that $c = \overline{a}$ shows the existential claim: given $u : \mathcal{M}$ with $\mathcal{M} \vDash u < \overline{n}$ we can conclude that $u$ must be a standard number $\overline{u}$. We then have the equivalences

$$\mathcal{M} \vDash \alpha(\overline{u}, b) \iff p\,u \iff \pi_u \mid a \iff \mathcal{M} \vDash \Pi(\overline{u}) \mid \overline{a}$$

since $a$ codes $p$ and $\Pi$ represents $\pi$. ◀

▶ **Lemma 26** (Infinite Coding in $\mathcal{M}$). *If* std *is stable,* $\mathcal{M} \not\cong \mathbb{N}$ *and* $\alpha(x)$ *a unary formula, we have*

$$\neg\neg\, \exists c : \mathcal{M}\ \forall u : \mathbb{N}.\ \ \mathcal{M} \vDash \alpha(\overline{u}) \leftrightarrow \Pi(\overline{u}) \mid c.$$

**Proof.** Using Lemma 25 for the present case where $\alpha$ is unary, we get

$$\mathcal{M} \vDash \neg\neg\, \exists\, c\, \forall\, u < \overline{n}.\ \alpha(u) \leftrightarrow \Pi(u) \mid c$$

for every $n : \mathbb{N}$, so by Lemma 22 (Overspill) we get

$$\begin{aligned} &\neg\neg\, \exists\, e > \mathbb{N}.\ \mathcal{M} \vDash \neg\neg\, \exists\, c\, \forall\, u < e.\ \alpha(u) \leftrightarrow \Pi(u) \mid c \\ \Longrightarrow\ &\neg\neg\, \exists c : \mathcal{M}\, \forall u : \mathbb{N}.\ \mathcal{M} \vDash \alpha(\overline{u}) \leftrightarrow \Pi(\overline{u}) \mid c. \end{aligned}$$

Where we used that given $\forall u : \mathcal{M} < e. (\dots)$ we can show $\forall u : \mathbb{N}. (\dots)$, since we have $e > \mathbb{N}$ and therefore $\overline{u} < e$ for any $u : \mathbb{N}$ by Fact 20. ◀

▶ **Lemma 27.** *If* std *is stable,* $\mathcal{M} \not\cong \mathbb{N}$ *and* $\mathcal{M} \vDash \alpha(\overline{u}, b)$ *is definite for every* $b : \mathcal{M}$, $u : \mathbb{N}$, *then we have*

$$\neg\neg\, \forall b : \mathcal{M}\ \exists c : \mathcal{M}\ \forall u : \mathbb{N}.\ \ \mathcal{M} \vDash \alpha(\overline{u}, b) \leftrightarrow \Pi(\overline{u}) \mid c.$$

**Proof.** Similar to the proof of Lemma 26, but we make use of the definiteness to get the stronger result out of Lemma 25 and then use Overspill to conclude. ◀

## 6 Church's Thesis for First-Order Arithmetic

Church's thesis (CT) [18, 40], states that every function $\mathbb{N} \to \mathbb{N}$ has a representation in a previously chosen, concrete model of computation. In the constructive type theory that we are working in, it is possible to consistently add CT as an axiom [42, 35, 8]. Given that we are treating computability in the context of HA, we choose a version of CT which uses a model of computation based on representing functions by formulas in the language of HA.

▶ **Axiom 28** ($\mathsf{CT}_\mathsf{Q}$). *For every function* $f : \mathbb{N} \to \mathbb{N}$ *there exists a binary* $\exists_1$ *formula* $\varphi_f(x, y)$ *such that for every* $n : \mathbb{N}$ *we have* $\mathsf{Q} \vdash \forall y.\, \varphi_f(\overline{n}, y) \leftrightarrow \overline{fn} = y$.

This formulation takes its justification from the standard result establishing the representability of $\mu$-recursive functions by $\Sigma_1$ formulae in Q [33, 26], combined with the fact that existential quantifiers can be compressed as mentioned in Section 3, to get the desired $\exists_1$ formula. We can now apply $\mathsf{CT}_\mathsf{Q}$ on the injective prime function $\pi$ to immediately settle Remark 24:

▶ **Fact 29.** *There is a binary formula representing the injective prime function* $\pi$ *in* Q.

Furthermore, we can use it to establish the representability of decidable and enumerable predicates in Q [30].

▶ **Definition 30.** *We call* $p : \mathbb{N} \to \mathbb{P}$ weakly representable *by* $\varphi_p(x)$ *if* $\forall n{:}\mathbb{N}.\, p\,n \leftrightarrow \mathsf{Q} \vdash \varphi_p(\overline{n})$*, and* strongly representable *if* $p\,n \to \mathsf{Q} \vdash \varphi_p(\overline{n})$ *and* $\neg p\,n \to \mathsf{Q} \vdash \neg\varphi_p(\overline{n})$ *for every* $n{:}\mathbb{N}$*.*

▶ **Lemma 31** (Representability Theorem (RT)). *Assume* $\mathsf{CT_Q}$*, and let* $p : \mathbb{N} \to \mathbb{P}$ *be given.*
1. *If* $p$ *is decidable, it is strongly representable by a unary* $\exists_1$ *formula.*
2. *If* $p$ *is enumerable, it is weakly representable by a unary* $\exists_2$ *formula.*

**Proof.** If $p$ is decidable, then there is a function $f : \mathbb{N} \to \mathbb{N}$ such that $\forall x{:}\mathbb{N}.\, p\,x \leftrightarrow f\,x = 0$ and by $\mathsf{CT_Q}$ there is a binary $\exists_1$ formula $\varphi_f(x, y)$ representing $f$. We then define $\varphi_p(x) := \varphi_f(x, \overline{0})$ and deduce

$$p\,n \implies f\,n = 0 \implies \mathsf{Q} \vdash \overline{f\,n} = \overline{0} \implies \mathsf{Q} \vdash \varphi_f(\overline{n}, \overline{0}) \implies \mathsf{Q} \vdash \varphi_p(\overline{n})$$
$$\neg p\,n \implies f\,n \neq 0 \implies \mathsf{Q} \vdash \neg(\overline{f\,n} = \overline{0}) \implies \mathsf{Q} \vdash \neg\varphi_f(\overline{n}, \overline{0}) \implies \mathsf{Q} \vdash \neg\varphi_p(\overline{n})$$

which shows that $p$ is strongly representable.

If $p$ is enumerable, then there is $f : \mathbb{N} \to \mathbb{N}$ such that $\forall x : \mathbb{N}.\, p\,x \leftrightarrow \exists n.\, f\,n = x + 1$ and by $\mathsf{CT_Q}$ there is a binary $\exists_1$ formula $\varphi_f(x, y)$ representing $f$. We then define $\varphi_p(x) := \exists\, n.\, \varphi_f(n, Sx)$ giving us

$$\mathsf{Q} \vdash \varphi_p(\overline{x}) \iff \mathsf{Q} \vdash \exists\, n.\, \varphi_f(n, S\overline{x}) \iff \exists n{:}\mathbb{N}.\, \mathsf{Q} \vdash \varphi_f(\overline{n}, S\overline{x})$$
$$\iff \exists n{:}\mathbb{N}.\, \mathsf{Q} \vdash \overline{f\,n} = S\overline{x} \iff \exists n{:}\mathbb{N}.\, f\,n = x + 1 \iff p\,x$$

which shows that $p$ is weakly representable by a $\exists_2$ formula. ◀

# 7 Tennenbaum's Theorem

We will now present several proofs of Tennenbaum's theorem, differing in the assumptions they make and the strength of their results. All of the proofs have in common that they start by the assumption $\mathcal{M} > \mathbb{N}$ to then make use of the coding lemma to encode a particular formula or predicate by an element of the model.

In Section 7.1 we will assume enumerability of the model, enabling a direct diagonal argument. This proof idea can be found in [3]. In Section 7.2 we look at the proof approach that is most prominently found in the literature [34, 14] and uses the existence of recursively inseparable sets. We sharpen this approach to a generalization only relying on decidability of the divisibility relation of the model.

Another variant of the usual proof was proposed in [20] and circumvents the usage of Overspill. In our constructive setting, this leads to a perceivable difference when it comes to the strength of the result. Lastly we look at the consequences of Tennenbaum's theorem, once the underlying semantics is made explicitly constructive. The latter two variations are discussed in Section 7.3.

## 7.1 Via a Diagonal Argument

We start by noting that every $\mathsf{HA}$ model can prove the most basic fact about divisibility.

▶ **Lemma 32** (Euclidean Lemma). *Given* $e, d{:}\mathcal{M}$ *we have*

$$\mathcal{M} \vDash \exists\, r\, q.\, e = q \cdot d + r \;\wedge\; (0 < d \to r < d)$$

*and the* uniqueness property *telling us that if* $r_1, r_2 < d$ *then* $q_1 \cdot d + r_1 = q_2 \cdot d + r_2$ *implies* $q_1 = q_2$ *and* $r_1 = r_2$*.*

**Proof.** For Euclid's lemma, there is a standard proof by induction on $e:\mathcal{M}$. The uniqueness claim requires some basic results about the order relation $<$. ◀

▶ **Lemma 33.** *If $\mathcal{M}$ is enumerable and discrete, then $\lambda nd.\, \mathcal{M} \vDash \overline{n} \mid d$ has a decider.*

**Proof.** Let $n:\mathbb{N}$ and $d:\mathcal{M}$ be given. By the Euclidean Lemma 32 we have $\exists q,r:\mathcal{M}.\, e = q\cdot d+r$. This existence is propositional, so presently we cannot use it to give a decision for $e \mid d$. Since $\mathcal{M}$ is enumerable, there is a surjective function $g : \mathbb{N} \to \mathcal{M}$ and the above existence therefore shows $\exists q,r:\mathbb{N}.\, e = (g\,q)\cdot d + (g\,r)$. Since equality is decidable in $\mathcal{M}$ and $\mathbb{N}^2$ is witnessing, we get $\Sigma q,r:\mathbb{N}.\, e = (g\,q)\cdot d + (g\,r)$, giving us computational access to $r$, now allowing us to construct the decision. By the uniqueness part of Lemma 32 we have $g\,r = 0 \leftrightarrow e \mid d$, so the decidability of $e \mid d$ is entailed by the decidability of $g\,r = 0$. ◀

▶ **Lemma 34.**
1. *If* std *is stable, then so is $\mathcal{M} \cong \mathbb{N}$.*
2. *Assuming* MP *and discreteness of $\mathcal{M}$, then* std *is stable.*

**Proof.** The first statement is trivial by Lemma 18. For the second, recall that $\mathsf{std}\, e$ stands for $\exists n:\mathbb{N}.\, \overline{n} = e$. Since $\overline{n} = e$ in $\mathcal{M}$ is decidable, stability follows from Fact 5. ◀

▶ **Lemma 35.** *If* std *is stable, $\mathcal{M} \ncong \mathbb{N}$, and $p:\mathbb{N} \to \mathbb{P}$ decidable, then potentially there is a code $c:\mathcal{M}$ such that $\forall n:\mathbb{N}.\; p\,n \leftrightarrow \mathcal{M} \vDash \overline{\pi_n} \mid c$.*

**Proof.** By RT, there is a formula $\varphi_p$ strongly representing $p$. Under the given assumptions, we can use the coding Lemma 26, yielding a code $c:\mathcal{M}$ for $\varphi_p$, such that $\forall u:\mathbb{N}.\, \mathcal{M} \vDash \varphi_p(\overline{u}) \leftrightarrow \Pi(\overline{u}) \mid c$. Overall this shows:

$$\begin{aligned} p\,n &\implies \mathsf{Q} \vdash \varphi_p(\overline{n}) \implies \mathcal{M} \vDash \varphi_p(\overline{n}) \implies \mathcal{M} \vDash \Pi(\overline{n}) \mid c \\ \neg p\,n &\implies \mathsf{Q} \vdash \neg\,\varphi_p(\overline{n}) \implies \neg\,\mathcal{M} \vDash \varphi_p(\overline{n}) \implies \neg\,\mathcal{M} \vDash \Pi(\overline{n}) \mid c. \end{aligned}$$

Since $p$ is decidable, the latter implication entails $\mathcal{M} \vDash \Pi(\overline{n}) \mid c \implies p\,n$, which overall shows the desired equivalence. ◀

This gives us the following version of Tennenbaum's theorem:

▶ **Theorem 36.** *Assuming* MP*, if $\mathcal{M}$ is enumerable and discrete, then $\mathcal{M} \cong \mathbb{N}$.*

**Proof.** By Lemma 34 it suffices to show $\neg\neg\mathcal{M} \cong \mathbb{N}$. So assume $\mathcal{M} \ncong \mathbb{N}$ and try to derive $\bot$. Given the enumerability, there is a surjective function $g:\mathbb{N} \to \mathcal{M}$. We use this to define the predicate $p := \lambda n:\mathbb{N}.\, \neg\,\mathcal{M} \vDash \overline{\pi_n} \mid g\,n$, which is decidable by Lemma 33. By Lemma 35 and surjectivity of $g$ then, there is some $c:\mathbb{N}$, such that $\neg\,\mathcal{M} \vDash \overline{\pi_c} \mid g\,c \overset{\text{def.}}{\iff} p\,c \overset{35}{\iff} \mathcal{M} \vDash \overline{\pi_c} \mid g\,c$ which gives the desired contradiction. ◀

## 7.2 Via Inseparable Predicates

The usual proof of Tennenbaum's theorem [14, 34] uses the existence of recursively inseparable sets and non-standard coding to establish the existence of a non-recursive set. In this situation, if we then were to again assume enumerability and discreteness of $\mathcal{M}$, we could easily reach the same conclusion as in Theorem 36. In the following however, we want to highlight that the proof which uses inseparable sets allows for a characterization of $\mathcal{M} \cong \mathbb{N}$ which only makes reference to the decidability of divisibility by numerals:

▶ **Definition 37.** *For $d:\mathcal{M}$ define the predicate $\overline{\cdot} \mid d := \lambda n:\mathbb{N}.\, \mathcal{M} \vDash \overline{n} \mid d$.*

So in particular, in the following we will not assume enumerability or discreteness of $\mathcal{M}$.

▶ **Definition 38.** *A pair $A, B : \mathbb{N} \to \mathbb{P}$ of predicates is called* inseparable *iff*

1. *they are disjoint, meaning $\forall n{:}\mathbb{N}.\, \neg(A\, n \wedge B\, n)$*
2. *there is no decidable $D : \mathbb{N} \to \mathbb{P}$ which includes $A$ i.e. $\forall n{:}\mathbb{N}.\, A\, n \to D\, n$ and is disjoint from $B$ i.e. $\forall n{:}\mathbb{N}.\, \neg(B\, n \wedge D\, n)$.*

▶ **Lemma 39.** *There are inseparable enumerable predicates $A, B : \mathbb{N} \to \mathbb{P}$.*

**Proof.** We use an enumeration $\Phi_n$ : $\mathsf{fm}$ of formulas to define disjoint predicates $A := \lambda n : \mathbb{N}.\, \mathsf{Q} \vdash \neg\, \Phi_n(\overline{n})$ and $B := \lambda n{:}\mathbb{N}.\, \mathsf{Q} \vdash \Phi_n(\overline{n})$. Since proofs over $\mathsf{Q}$ can be enumerated, $A$ and $B$ are enumerable. Assume we are given a decidable predicate $D$ which includes $A$ and is disjoint from $B$. Using $\mathsf{RT}$ and the enumeration, there is $d{:}\mathbb{N}$ such that $\Phi_d$ strongly represents $D$. This gives us $D\, d \implies \mathsf{Q} \vdash \Phi_d(\overline{d}) \implies B\, d$, contradicting the disjointness of $B$ and $D$, therefore showing $\neg D\, d$. Furthermore, representability gives us $\neg D\, d \implies \mathsf{Q} \vdash \neg\Phi_d(\overline{d}) \implies A\, d$ and since $A$ is included in $D$, this shows $\neg D\, d \implies D\, d$. Overall this gives us a contradiction. ◀

▶ **Corollary 40.** *There is a pair $\alpha(z), \beta(z)$ of unary $\exists_2$ formulas such that $A{:=}\lambda n{:}\mathbb{N}.\, \mathsf{Q} \vdash \alpha(\overline{n})$ and $B := \lambda n{:}\mathbb{N}.\, \mathsf{Q} \vdash \beta(\overline{n})$ are inseparable and enumerable.*

**Proof.** We get the desired formulas by using the weak representability of Lemma 31 on the predicates given by Lemma 39. ◀

▶ **Lemma 41.** *Assuming stability of $\mathsf{std}$ and $\mathcal{M} \ncong \mathbb{N}$, then $\neg\neg\, \exists d{:}\mathcal{M}.\, \neg\mathsf{Dec}(\overline{\cdot} \mid d)$.*

**Proof.** By Corollary 40 there are inseparable formulas $\exists x, y.\, \alpha_0(x, y, z)$ and $\exists x, y.\, \beta_0(x, y, \overline{n})$ such that $\alpha_0, \beta_0$ are $\Delta_1$. Since they are disjoint, we have:

$$\mathbb{N} \vDash \forall\, x\, y\, u\, v\, z < \overline{n}.\, \neg(\alpha_0(x, y, z) \wedge \beta_0(u, v, z))$$

for every bound $n{:}\mathbb{N}$. By Lemma 13 we then get

$$\mathcal{M} \vDash \forall\, x\, y\, u\, v\, z < \overline{n}.\, \neg(\alpha_0(x, y, z) \wedge \beta_0(u, v, z))$$

and using Overspill we therefore potentially have $e{:}\mathcal{M}$ with

$$\mathcal{M} \vDash \forall\, x\, y\, u\, v\, z < e.\, \neg(\alpha_0(x, y, z) \wedge \beta_0(u, v, z))$$

showing the disjointness of $\alpha_0, \beta_0$ when everything is bounded by $e$. We now define the predicate $X := \lambda n{:}\mathbb{N}.\, \mathcal{M} \vDash \exists\, x, y < e.\, \alpha_0(x, y, \overline{n})$ and note that

- If $\mathsf{Q} \vdash \exists\, x, y.\, \alpha_0(x, y, \overline{n})$ there are $m_1, m_2$ with $\mathbb{N} \vDash \alpha_0(\overline{m_1}, \overline{m_2}, \overline{n})$ and $\mathcal{M} \vDash \alpha_0(\overline{m_1}, \overline{m_2}, \overline{n})$ by Lemma 13. We therefore get $Xn$.
- Assume that $Xn \wedge \mathsf{Q} \vdash \exists\, x, y.\, \beta_0(x, y, \overline{n})$. Then similarly to above, there are $m_1, m_2 : \mathbb{N}$ with $\mathcal{M} \vDash \beta_0(\overline{m_1}, \overline{m_2}, \overline{n})$, showing $\mathcal{M} \vDash \exists\, x, y < e.\, \beta_0(x, y, \overline{n})$. Together with $Xn$ this contradicts the disjointness of $\alpha_0, \beta_0$ under the bound $e$.

Due to the inseparability of the given formulas, this shows that $X$ cannot be decidable and by Lemma 27 there is now potentially a code $d{:}\mathcal{M}$ with $Xn \Leftrightarrow \mathcal{M} \vDash \overline{\pi_n} \mid d$. ◀

▶ **Fact 42.** *For every $e{:}\mathcal{M}$ we have $\mathsf{std}\, e \to \mathsf{Dec}(\overline{\cdot} \mid e)$.*

▶ **Corollary 43.** *Given $\mathsf{MP}$ and discrete $\mathcal{M}$, we have $\mathcal{M} \cong \mathbb{N}$ iff $\forall d{:}\mathcal{M}.\, \neg\neg\mathsf{Dec}(\overline{\cdot} \mid d)$.*

**Proof.** The first implication follows by Fact 42. For the converse, note that the contraposition of Lemma 41 shows $\forall d{:}\mathcal{M}.\, \neg\neg\mathsf{Dec}(\overline{\cdot} \mid d) \to \neg\neg\mathcal{M} \cong \mathbb{N}$ where the conclusion is equivalent to $\mathcal{M} \cong \mathbb{N}$ due to Lemma 34. ◀

### 7.3 Variants of the Theorem

We now investigate two further variants of the theorem, by making two further assumptions: the existence of formulas which satisfy a stronger notion of inseparability and that the coding lemma can be proven inside of HA.

▶ **Definition 44.** *Two formulas $\alpha(x), \beta(x)$ are called* HA-inseparable *if $\lambda n{:}\mathbb{N}.\, \mathsf{Q} \vdash \alpha(\overline{n})$ and $\lambda n{:}\mathbb{N}.\, \mathsf{Q} \vdash \beta(\overline{n})$ are inseparable and one can also show $\mathsf{HA} \vdash \neg\exists\, x.\, \alpha(x) \land \beta(x)$.*

▶ **Hypothesis 1.** *There are $\Delta_1$ formulas $\alpha_0, \beta_0$ such that $\exists z.\, \alpha_0(z, x)$ and $\exists z.\, \beta_0(z, x)$ are* HA*-inseparable.*

▶ **Hypothesis 2.** *For any binary $\Delta_1$ formula $\varphi(x, y)$,* HA *can prove the following coding lemma: $\mathsf{HA} \vdash \forall\, n\, b\, \exists\, c\, \forall\, u < n.\ (\exists z < b.\, \varphi(z, u)) \leftrightarrow \Pi(u) \mid c$.*

According to [24], one way of establishing Hypothesis 1 is by taking the construction of inseparable formulas as seen earlier, and internalizing the given proof within HA. Similarly, Hypothesis 2 is justified by noting that its proof should be an internalized version of the proof of Lemma 23.

The following variant of Tennenbaum's theorem is based on an observation by Makholm [20]. Most importantly, it avoids the usage of Overspill, by using Hypothesis 2. In contrast to the result in Section 7.1 we want to highlight that the next theorem does not presuppose MP or the stability of std.

▶ **Theorem 45** (Makholm). *We have $\mathcal{M} > \mathbb{N}$ if and only if $\exists d{:}\mathcal{M}. \neg\mathsf{Dec}(\bar{\cdot} \mid d)$.*

**Proof.** First note that the converse follows from Fact 42. Now assume we have $e{:}\mathcal{M} > \mathbb{N}$. By Hypothesis 1 there are HA-inseparable $\exists_1$ formulas $\exists z.\, \alpha_0(z, x)$ and $\exists z.\, \beta_0(z, x)$, where $\alpha_0, \beta_0$ are binary $\Delta_1$ formulas. Then let $X := \lambda n{:}\mathbb{N}.\, \mathcal{M} \vDash \exists\, z < e.\, \alpha_0(z, \overline{n})$.

- If $\mathsf{Q} \vdash \exists\, z.\, \alpha_0(z, \overline{n})$ there is $m{:}\mathbb{N}$ with $\mathbb{N} \vDash \alpha_0(\overline{m}, \overline{n})$ and $\mathcal{M} \vDash \alpha_0(\overline{m}, \overline{n})$ by Lemma 13. We therefore get $Xn$.
- Assuming $Xn \land \mathsf{Q} \vdash \exists\, z.\, \beta_0(z, \overline{n})$, then similarly to above, there is $m{:}\mathbb{N}$ with $\mathcal{M} \vDash \beta_0(\overline{m}, \overline{n})$, showing $\mathcal{M} \vDash \exists\, z < e.\, \beta_0(z, \overline{m})$. But together with $Xn$ this contradicts the deductive disjointness property of the HA-inseparable formulas $\alpha_0$ and $\beta_0$.

Due to the inseparability of the given $\exists_1$ formulas, this shows that $X$ is not decidable. Using soundness on Hypothesis 2 for $\varphi := \alpha_0$ and $n, b := e$, we get

$$\mathcal{M} \vDash \exists\, c\, \forall\, u < e.\ \big(\exists\, z < e.\, \alpha_0(z, u)\big) \leftrightarrow \Pi(u) \mid c.$$

So there is a code $c{:}\mathcal{M}$ such that $X$ is coded by it, showing that $\bar{\cdot} \mid c$ cannot be decidable. ◀

▶ **Corollary 46.** *We have $\forall e{:}\mathcal{M}. \neg\neg\,\mathsf{std}\, e$ iff $\forall d{:}\mathcal{M}. \neg\neg\mathsf{Dec}(\bar{\cdot} \mid d)$.*

McCarty [24, 23] considered Tennenbaum's theorem with constructive semantics. Instead of models placed in classical set theory, he assumes an intuitionistic theory (e.g. IZF), making the interpretation of the object-level disjunction much stronger. We simulate this in our type theory by assuming the following choice principle:

▶ **Definition 47.** *By* AUC *we denote the principle of unique choice:*

$$\forall X\, Y\, R.\, (\forall x\, \exists! y.\, Rxy) \to \exists f : X \to Y. \forall x.\, Rx(fx)$$

Note that CT and AUC combined prove the negation of LEM [7]. In the following, we are therefore (deliberately) anti-classical.

▶ **Lemma 48.** *For any formula $\varphi(x,y)$ we have $\mathcal{M} \vDash \forall b.\, \neg\neg\forall x, y < b.\, \varphi(x,y) \lor \neg\varphi(x,y)$.*

**Proof.** Single instances of the law of excluded middle are provable under double negation. We can then use this in combination with an induction on the bound $b$ to prove the claim. ◀

▶ **Lemma 49.** *Assuming* AUC *and $\mathcal{M} > \mathbb{N}$, we have $\forall d{:}\mathcal{M}. \neg\neg\mathsf{Dec}(\overline{\cdot} \mid d)$.*

**Proof.** Let $d{:}\mathcal{M}$ be given and assume $e{:}\mathcal{M} > \mathbb{N}$. Then we have $e + d + 1 > \mathbb{N}$ and using Lemma 48 we get

$$\begin{aligned} & \mathcal{M} \vDash \forall b.\, \neg\neg\forall x, y < b.\, \varphi(x,y) \lor \neg\varphi(x,y) \\ \Longrightarrow\ & \neg\neg\, \mathcal{M} \vDash \forall x, y < (e + d + 1).\ \varphi(x,y) \lor \neg\varphi(x,y) \\ \Longrightarrow\ & \neg\neg\forall n{:}\mathbb{N}.\, \mathcal{M} \vDash \varphi(\overline{n}, d) \lor \neg\varphi(\overline{n}, d) \\ \Longrightarrow\ & \neg\neg\forall n{:}\mathbb{N}.\, \mathcal{M} \vDash \varphi(\overline{n}, d) + \neg\, \mathcal{M} \vDash \varphi(\overline{n}, d) \end{aligned}$$

where the last implication is possible, since AUC implies the decidability of definite propositions. For the choice $\varphi(x,y) := x \mid y$ we then get the desired result. ◀

▶ **Corollary 50.** *Assuming* AUC*, then there are no non-standard models.*

**Proof.** Given $\mathcal{M} > \mathbb{N}$, Lemma 49 entails $\neg\exists d : \mathcal{M}. \neg\mathsf{Dec}(\overline{\cdot} \mid d)$, in contradiction to Theorem 45. ◀

Still assuming both Hypothesis 1 and Hypothesis 2 we can then derive:

▶ **Corollary 51** (McCarty)**.** *Given* AUC *and* MP*,* HA *is categorical.*

**Proof.** Given that $\mathsf{HA} \vdash \forall xy.\, x = y \lor \neg\, x = y$, AUC entails that every model $\mathcal{M} \vDash \mathsf{HA}$ is discrete, showing the stability of std by Lemma 34. Combined with Corollary 50 this shows $\mathcal{M} \cong \mathbb{N}$. ◀

# 8 Discussion

## 8.1 General Remarks

In Section 7, we presented several proofs of Tennenbaum's theorem which we summarize in the below table, listing their assumptions[7] on the left and the conclusion on the right.

| MP | AUC | discrete | HA-insep. | Conclusion | from |
|---|---|---|---|---|---|
| • | | • | | $\mathbb{N} \cong \mathcal{M}$ iff $\mathcal{M}$ enumerable | Theorem 36 |
| • | | • | | $\mathcal{M} > \mathbb{N} \to \neg\neg\exists d. \neg\mathsf{Dec}(\overline{\cdot} \mid d)$ | Lemma 41 |
| | | | • | $\mathcal{M} > \mathbb{N} \leftrightarrow \exists d. \neg\mathsf{Dec}(\overline{\cdot} \mid d)$ | Theorem 45 |
| • | • | | • | $\mathbb{N} \cong \mathcal{M}$ | Corollary 51 |

First note that since HA can show definiteness of equality, the above listed assumption of the model $\mathcal{M}$ being discrete is equivalent to $\mathcal{M}$ being separated.

Comparing Theorem 45 to Theorem 36 and Lemma 41 we see that its conclusion is constructively stronger. The noteworthy observation about Theorem 45 is that it cannot be reached by the proofs given in Section 7.2, as they crucially depend on Overspill and therefore MP and discreteness. The result only becomes possible once we use a stronger notion of inseparability for formulas and avoid the usage of Overspill.

[7] We do not list the global assumption $\mathsf{CT_Q}$. Both Hypothesis 1 and Hypothesis 2 are provable but left unmechanized in Coq, we only list the former to highlight where it was used.

As was pointed out by McCarty in [24], a weaker version WCT of CT suffices for his proof, where the code representing a given function is hidden behind a double negation. He mentions in [25] that WCT is still consistent with the Fan theorem, while CT is not. Analogously, the following weakening of $\mathsf{CT_Q}$ suffices for all of the proofs that we have presented:

▶ **Definition 52** ($\mathsf{WCT_Q}$). *For every function $f : \mathbb{N} \to \mathbb{N}$ there* potentially *is a binary $\exists_1$ formula $\varphi_f(x, y)$ such that for every $n : \mathbb{N}$ we have $\mathsf{Q} \vdash \forall y.\, \varphi_f(\overline{n}, y) \leftrightarrow \overline{fn} = y$.*

This only needs few changes of the presented proofs and we verified this in the Coq project.[8] An advantage of $\mathsf{WCT_Q}$ over $\mathsf{CT_Q}$ is that the former follows from the double negation of the latter and is therefore negative, ensuring that its assumption does not block computation [5].

Depending on the fragment of first-order logic one can give constructive proofs of the model existence theorem [10], producing a countable syntactic model with computable functions for every consistent theory. By the argument given in the introduction, model existence would yield a countable and computable non-standard model of PA, which at first glance seems to contradict the statement of Tennenbaum's theorem. For any countable non-standard model of PA however, Theorem 45 and Lemma 33 entail that neither equality nor apartness can be decidable. This is similar in spirit to the results in [36], showing that even if the functions of the model are computable, non-computable behavior still emerges, but in relation to equality.

## 8.2 Coq Mechanization

The Coq development is axiom-free and the usage of crucial but constructively justified axioms $\mathsf{CT_Q}$ and MP are localized in the relevant sections. Apart from these, there are the two facts in Section 7.3 we have labeled as hypotheses, and which were taken as additional assumptions in the relevant sections. They are expected to be provable and would on paper usually be treated as facts and simply used, but since our treatment is backed up by a mechanization, we prefer to make these assumptions very explicit in the accompanying text.

In total, the development counts roughly 5400 lines of code. From those, 3000 loc on the specification of first-order logic and basic results about PA models were reused from earlier work [9, 10, 16, 15]. Notably, the formalization of the various coding lemmas from Section 5 took 460 loc and all variants of Tennenbaum's theorem come to a total of only 860 lines.

In contrast to the previous developments, where equality was treated as a relation symbol, we decided to treat equality as a primitive of the syntax. This is chosen as a mere simplification to ease working in an abstract model and is expected to be straightforward to eliminate.

## 8.3 Related Work

Presentations of first-order logic in the context of proof-checking have already been discussed and used, among others, by Shankar [32], Paulson [28], O'Connor [26], as well as Han and van Doorn [12]. The particular mechanization of first-order logic we use is based on several previous projects [9, 10, 16, 15] and part of the Coq library of undecidability proofs [11].

Classical proofs of Tennenbaum's theorem can be found in [3, 34, 14]. There are also refinements of the theorem which show that computability of either operation suffices [22] as well as a weaker induction scheme [41, 4]. Constructive accounts were given by McCarty [23,

[8] We could have presented all of the results with respect to $\mathsf{WCT_Q}$. We opted against this in favor of $\mathsf{CT_Q}$, to avoid additional handling of double negations and to keep the proofs more readable.

24] and Plisko [29], and a relatively recent investigation into Tennenbaum phenomena was conducted by Godziszewski and Hamkins [36].

Synthetic computability theory was introduced by Richman and Bauer [31, 1] and initially applied to constructive type theory by Forster, Kirst, and Smolka [9]. Their synthetic approach to undecidability results has been used in several other projects, all merged into the Coq library of undecidability proofs [11].

For an account of CT as an axiom in constructive mathematics we refer to Kreisel [18] and Troelstra [39]. Investigations into CT and its connections to other axioms of synthetic computability based on constructive type theory were done by Forster [7, 8].

## 8.4 Future Work

We would like to give a proper formalization of the arithmetical hierarchy, in particular implementing our semantic treatment of $\Delta_1$ formulas with a syntactic restriction to formulas with bounded quantification. For a suitable definition, it could then also be shown that every $\Sigma_1$ formula is $\exists_1$, making the treatment of $\mathsf{CT_Q}$ and RT more uniform. A definition of the full hierarchy would allow us to conduct an analysis concerning the arithmetical strength of the induction scheme needed to establish Tennenbaum's theorem.

We would like to further justify $\mathsf{CT_Q}$ by starting off with the more conventional formulation of CT for some canonical model of computation, as for instance stated in [7], and verifying that it yields $\mathsf{CT_Q}$. This should be in reach by connecting the mechanization of the DPRM theorem, given by Larchey-Wendling and Forster [19], with its reduction to Q, given by Kirst and Hermes [15].

We plan to mechanize the facts left informal in Section 7.3, namely Hypothesis 1 and Hypothesis 2. As these require sizeable syntactic derivations inside of HA but are not so central for the main result, we decided to avoid the necessary cumbersome manipulations in Coq. Their mechanization could possibly benefit from the proof mode developed in [13].

A more satisfying rendering of McCarty's result will be achieved by changing Definition 9, putting the interpretations of formulas on the (proof-relevant) type level instead of the propositional level, therefore removing the need to assume AUC to break the barrier from the propositional to the type level.

Following usual practice in textbooks, in Coq we consider equality a syntactic primitive and only regard models interpreting it as actual equality. When treated as axiomatized relation instead, we could consider the (slightly harder to work with) setoid models and obtain the more general result that no computable non-standard setoid model exists.

The presented versions of Tennenbaum's theorem do not explicitly mention the computability of addition or multiplication of the model, and as mentioned in Section 1 this is due to the chosen synthetic approach. To make these assumptions explicit again, we could assume an abstract version of CT which makes reference to a $T$ predicate [17, 7], and expresses that every $T$-computable function is representable in Q. We can then then distinguish between addition or multiplication being $T$-computable and formalize the result that $T$-computability of either operation leads to the model being standard [22].

## References


1 Andrej Bauer. First steps in synthetic computability theory. *Electronic Notes in Theoretical Computer Science*, 155:5–31, 2006.

2 Andrej Bauer. Intuitionistic mathematics for physics, 2008. URL: `http://math.andrej.com/2008/08/13/intuitionistic-mathematics-for-physics/`.

3 George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. Cambridge university press, 2002.

4 Patrick Cegielski, Kenneth McAloon, and George Wilmers. Modèles récursivement saturés de l'addition et de la multiplication des entiers naturels. In *Studies in Logic and the Foundations of Mathematics*, volume 108, pages 57–68. Elsevier, 1982.

5 Thierry Coquand, Nils A. Danielsson, Martın H. Escardó, Ulf Norell, and Chuangjie Xu. Negative consistent axioms can be postulated without loss of canonicity. *Unpublished note*, 2013. URL: `https://www.cs.bham.ac.uk/~mhe/papers/negative-axioms.pdf`.

6 Thierry Coquand and Gérard Huet. The calculus of constructions. Technical report, INRIA, 1986.

7 Yannick Forster. Church's Thesis and Related Axioms in Coq's Type Theory. 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 21:1–21:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CSL.2021.21`.

8 Yannick Forster. Parametric Church's Thesis: Synthetic Computability Without Choice. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 70–89, Cham, 2022. Springer International Publishing.

9 Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51, 2019.

10 Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation*, 31(1):112–151, 2021.

11 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 The Sixth International Workshop on Coq for Programming Languages*, 2020.

12 Jesse M. Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 353–366, 2020.

13 Johannes Hostert, Mark Koch, and Dominik Kirst. A toolbox for mechanised first-order logic. In *The Coq Workshop 2021*, 2021.

14 Richard Kaye. Tennenbaum's theorem for models of arithmetic. *Set Theory, Arithmetic, and Foundations of Mathematics. Ed. by J. Kennedy and R. Kossak. Lecture Notes in Logic. Cambridge*, pages 66–79, 2011.

15 Dominik Kirst and Marc Hermes. Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ITP.2021.23`.

16 Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's theorem in Coq: A Constructive Approach to Finite Model Theory. In *International Joint Conference on Automated Reasoning*, pages 79–96. Springer, 2020.

17 Stephen C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.

18 Georg Kreisel. Church's thesis: a kind of reducibility axiom for constructive mathematics. In *Studies in Logic and the Foundations of Mathematics*, volume 60, pages 121–150. Elsevier, 1970.

19 Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019*, volume 131, pages 27–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

20 Henning Makholm. Tennenbaum's theorem without overspill. Mathematics Stack Exchange. (version: 2014-01-24). URL: `https://math.stackexchange.com/q/649457`.

21 Bassel Mannaa and Thierry Coquand. The independence of Markov's principle in type theory. *Logical Methods in Computer Science*, 13, 2017.

22 Kenneth McAloon. On the complexity of models of arithmetic. *The Journal of Symbolic Logic*, 47(2):403–415, 1982.

23 David C. McCarty. Variations on a thesis: intuitionism and computability. *Notre Dame Journal of Formal Logic*, 28(4):536–580, 1987.

24 David C. McCarty. Constructive validity is nonarithmetic. *The Journal of Symbolic Logic*, 53(4):1036–1041, 1988. URL: `http://www.jstor.org/stable/2274603`.

25 David C. McCarty. Incompleteness in intuitionistic Metamathematics. *Notre Dame journal of formal logic*, 32(3):323–358, 1991.

26 Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 245–260. Springer, 2005.

27 Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

28 Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

29 Valerii E. Plisko. Constructive formalization of the Tennenbaum theorem and its applications. *Mathematical notes of the Academy of Sciences of the USSR*, 48(3):950–957, 1990.

30 Panu Raatikainen. Gödel's Incompleteness Theorems. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2021 edition, 2021.

31 Fred Richman. Church's thesis without tears. *The Journal of symbolic logic*, 48(3):797–803, 1983.

32 Natarajan Shankar. *Proof-checking metamathematics (theorem-proving)*. PhD thesis, The University of Texas at Austin, 1986.

33 Peter Smith. *An introduction to Gödel's theorems*. Cambridge University Press, 2013.

34 Peter Smith. Tennenbaum's theorem. Technical report, Cambridge University, 2014. URL: `https://www.logicmatters.net/resources/pdfs/tennenbaum_new.pdf`.

35 Andrew Swan and Taichi Uemura. On Church's Thesis in Cubical Assemblies, 2019. `arXiv:1905.03014`.

36 Michał T. Godziszewski and Joel D. Hamkins. Computable quotient presentations of models of arithmetic and set theory. *arXiv e-prints*, pages arXiv–1702, 2017.

37 The Coq Development Team. The Coq Proof Assistant, January 2022. `doi:10.5281/zenodo.5846982`.

38 Stanley Tennenbaum. Non-archimedean models for arithmetic. *Notices of the American Mathematical Society*, 6(270):44, 1959.

39 Anne S. Troelstra. *Metamathematical investigation of intuitionistic arithmetic and analysis*, volume 344. Springer Science & Business Media, 1973.

40 Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics, Vol. 1. Studies in Logic and the Foundations of Mathematics, Vol. 121*. North-Holland Press, Amsterdam, 1988.

41 George Wilmers. Bounded existential induction. *The Journal of Symbolic Logic*, 50(1):72–90, 1985.

42 Norihiro Yamada. Game semantics of Martin-Löf type theory, part III: its consistency with Church's thesis, 2020. `arXiv:2007.08094`.

# A Deduction Systems

Intuitionistic natural deduction $\vdash : \mathsf{List}(\mathsf{fm}) \to \mathsf{fm} \to \mathbb{P}$ is defined inductively by the rules

$$\frac{\varphi \in \Gamma}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \varphi}$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma \vdash \varphi \vee \psi \quad \Gamma, \varphi \vdash \theta \quad \Gamma, \psi \vdash \theta}{\Gamma \vdash \theta}$$

$$\frac{\Gamma[\uparrow] \vdash \varphi}{\Gamma \vdash \forall \varphi} \qquad \frac{\Gamma \vdash \forall \varphi}{\Gamma \vdash \varphi[t]} \qquad \frac{\Gamma \vdash \varphi[t]}{\Gamma \vdash \exists \varphi} \qquad \frac{\Gamma \vdash \exists \varphi \quad \Gamma[\uparrow], \varphi \vdash \psi[\uparrow]}{\Gamma \vdash \psi}$$

where we get the classical variant $\vdash_c$ by adding Peirce's rule as an axiom:

$$\frac{}{\Gamma \vdash_c ((\varphi \to \psi) \to \varphi) \to \varphi}$$

The deduction systems lift to possibly infinite contexts $\mathcal{T} : \mathsf{fm} \to \mathbb{P}$ by writing $\mathcal{T} \vdash \varphi$ if there is a finite $\Gamma \subseteq \mathcal{T}$ with $\Gamma \vdash \varphi$.

# Constructing Unprejudiced Extensional Type Theories with Choices via Modalities

**Liron Cohen**
Ben-Gurion University of the Negev, Beer-Sheva, Israel

**Vincent Rahli**
University of Birmingham, UK

## Abstract

Time-progressing expressions, i.e., expressions that compute to different values over time such as Brouwerian choice sequences or reference cells, are a common feature in many frameworks. For type theories to support such elements, they usually employ sheaf models. In this paper, we provide a general framework in the form of an extensional type theory incorporating various time-progressing elements along with a general possible-worlds forcing interpretation parameterized by modalities. The modalities can, in turn, be instantiated with topological spaces of bars, leading to a general sheaf model. This parameterized construction allows us to capture a distinction between theories that are "agnostic", i.e., compatible with classical reasoning in the sense that classical axioms can be validated, and those that are "intuitionistic", i.e., incompatible with classical reasoning in the sense that classical axioms can be proven false. This distinction is made via properties of the modalities selected to model the theory and consequently via the space of bars instantiating the modalities. We further identify a class of time-progressing elements that allows deriving "intuitionistic" theories that include not only choice sequences but also simpler operators, namely reference cells.

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

**Keywords and phrases** Intuitionism, Extensional Type Theory, Constructive Type Theory, Realizability, Choice sequences, References, Classical Logic, Theorem proving, Agda

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.10

**Funding** This research was partially supported by Grant No. 2020145 from the United States-Israel Binational Science Foundation (BSF).

## 1 Introduction

Time-progressing elements are a common feature in many frameworks. These are elements whose value can change over time. Examples include mutable reference cells which are pervasive in programming languages, and free-choice sequences which are key components in logical systems such as Brouwer's intuitionistic logic [24, 40, 39, 38, 26, 43, 30]. A free-choice sequence is a primitive concept of a sequence that is never complete and can always be extended over time, and whose choices are allowed to be made freely, i.e., not generated by a predefined procedure. Capturing the non-deterministic, time-progressing behavior of such elements in a formal setting often relies on sheaf models, which logical formulas can interact with through a forcing interpretation, e.g., [19, 42].

The inclusion of such elements in a logical system has far reaching consequences. In particular, many works have used the existence of choice-sequences to show incompatibility with classical reasoning. For example, Kripke's Schema, which relies on the notion of choice sequences, is inconsistent with Church's Thesis [41, Sec.5]. They have also been used to refute classical results such as *"any real number different from* $0$ *is also apart from* $0$*"* [22, Ch.8]. Similarly, a weak counterexample of the Law of Excluded Middle (LEM) was provided by defining a choice sequence of numbers in which the value 1 can only be picked once an

© Liron Cohen and Vincent Rahli;
licensed under Creative Commons License CC-BY 4.0

undecided conjecture has been resolved (proved or disproved), and then by showing that one could resolve this undecided conjecture using LEM [8, Ch.1,Sec.1]. Kripke [29, Sec.1.1] also used choice sequences to refute other classical results, namely Kuroda's conjecture and Markov's Principle (MP) in Kreisel's FC system [25]. This technique was later generalized using sheaf models [19, 42] to refute classical axioms. For example, in [14] the independence of MP with Martin-Löf's type theory was proven using a forcing method where the forcing conditions capture the unconstrained nature of free-choice sequences in Kripke's proof. However, using a concrete sheaf model, it was shown in [5] that choice sequences can be made compatible with classical reasoning. This was however done by committing to a particular model, disabling the ability to derive "purely" intuitionistic theories.

This paper goes one step further by providing a general framework in the form of an extensional type theory that incorporates a notion of time progression through a Kripke frame, as well as elements that progress over time. The framework uses a general possible-worlds forcing interpretation parameterized by a modality, which, in turn can be instantiated with topological spaces of bars, leading to a general sheaf model. Thus, our generic type theory, denoted by $\mathrm{TT}_{\mathcal{C}}^{\square}$, is modeled through an abstract modality $\square$ and is parameterized by a type of time-progressing choice operators $\mathcal{C}$, which can both be instantiated to derive theories that are either compatible or incompatible with classical logic. $\mathrm{TT}_{\mathcal{C}}^{\square}$'s syntax and operational semantics are presented by first describing its time-independent core in Sec. 2.2, and then its time-progressing components in Sec. 3. In particular, $\mathrm{TT}_{\mathcal{C}}^{\square}$ can be instantiated with different choice operators described in Sec. 3.2. $\mathrm{TT}_{\mathcal{C}}^{\square}$'s inference rules are standard and are presented in Appx. A. They reflect the semantics of the types, which are given meaning through a forcing interpretation [10, 11, 3]parameterized by a modality $\square$ presented in Sec. 4.

We call $\mathrm{TT}_{\mathcal{C}}^{\square}$ an "unprejudiced" type theory since we can tune the parameters to obtain theories that are either "agnostic", i.e., compatible with classical reasoning (in the sense that classical axioms can be validated), or that are "intuitionistic", i.e., incompatible with classical reasoning (in the sense that classical axioms can be proven false). Concretely, we identify classes of choice operators and modalities that are sufficient to derive the negation of classical axioms, as well as classes that are sufficient to validate classical axioms in Sec. 5. We further show that $\mathrm{TT}_{\mathcal{C}}^{\square}$ can be validated w.r.t. standard sheaf models in Sec. 6, which presents classes of sheaf models over topological spaces of bars that are used to instantiate the modalities. We provide examples of classes of bar spaces $B$ and choice operators $\mathcal{C}$ that allow proving the consistency of $\mathrm{TT}_{\mathcal{C}}^{B}$ with LEM, and classes that allow proving the consistency of $\mathrm{TT}_{\mathcal{C}}^{B}$ with the negation of classical axioms such as LEM. In particular, we show that even though choice sequences can be used to validate the negation of classical axioms, they are not necessary, and in fact much simpler choice operators, e.g. mutable references, are enough.

# 2 Background

## 2.1 Metatheory

Our metatheory is Agda's type theory [1]. The results presented in this paper have been formalized in Agda, and the formalization is available here: `https://github.com/vrahli/opentt/`. We use $\forall, \exists, \wedge, \vee, \rightarrow, \neg$ in place of Agda's logical connectives in this paper. Agda provides an hierarchy of types annotated with universe labels which we omit for simplicity. Following Agda's terminology, we refer to an Agda type as a *set*, and reserve the term *type* for $\mathrm{TT}_{\mathcal{C}}^{\square}$'s types. We use $\mathbb{P}$ as the type of sets that denote propositions; $\mathbb{N}$ for the set of natural numbers; and $\mathbb{B}$ for the set of Booleans `true` and `false`. We use induction-recursion to define the forcing interpretation in Sec. 4, where we use function extensionality to interpret universes.

$$
\begin{array}{llllllll}
v \in \mathsf{Value} ::= vt & \text{(type)} & \mid \lambda x.t & \text{(lambda)} & \mid \star & \text{(constant)} \\
\quad \mid \underline{n} & \text{(number)} & \mid \mathtt{inl}(t) & \text{(left injection)} & \mid \delta & \text{(choice name)} \\
\quad \mid \langle t_1, t_2 \rangle & \text{(pair)} & \mid \mathtt{inr}(t) & \text{(right injection)} & & \\
vt \in \mathsf{Type} ::= \mathbf{\Pi} x{:}t_1.t_2 & \text{(product)} & \mid \{x : t_1 \mid t_2\} & \text{(set)} & \mid t_1 + t_2 & \text{(disjoint union)} \\
\quad \mid \mathbf{\Sigma} x{:}t_1.t_2 & \text{(sum)} & \mid t_1 = t_2 \in t & \text{(equality)} & \mid \wr t & \text{(time truncation)} \\
\quad \mid \mathbb{U}_i & \text{(universe)} & \mid \mathsf{Nat} & \text{(numbers)} & & \\
t \in \mathsf{Term} ::= x & \text{(variable)} & \mid t_1 \; t_2 & & & \text{(application)} \\
\quad \mid v & \text{(value)} & \mid \mathtt{let}\ x, y = t_1\ \mathtt{in}\ t_2 & & & \text{(pair destructor)} \\
\quad \mid \mathtt{fix}(t) & \text{(fixpoint)} & \mid \mathtt{case}\ t\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t_1 \mid \mathtt{inr}(y) \Rightarrow t_2 & & & \text{(injection destructor)}
\end{array}
$$

$$
\begin{array}{ll}
(\lambda x.t)\ u \mapsto_{w} t[x \backslash u] & \mathtt{let}\ x, y = \langle t_1, t_2 \rangle\ \mathtt{in}\ t \mapsto_{w} t[x \backslash t_1; y \backslash t_2] \\
\mathtt{fix}(v) \mapsto_{w} v\ \mathtt{fix}(v) & \mathtt{case}\ \mathtt{inl}(t)\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t_1 \mid \mathtt{inr}(y) \Rightarrow t_2 \mapsto_{w} t_1[x \backslash t] \\
\delta(\underline{n}) \mapsto_{w} \mathsf{choice?}(w, \delta, n) & \mathtt{case}\ \mathtt{inr}(t)\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t_1 \mid \mathtt{inr}(y) \Rightarrow t_2 \mapsto_{w} t_2[y \backslash t]
\end{array}
$$

**Figure 1** Core syntax (above) and small-step operational semantics (below).

We do not discuss this further here and the interested reader is referred to forcing.lagda in the Agda code for further details. Classical reasoning is only used once in Lem. 20 to establish the compatibility of instances of $\mathrm{TT}_{\mathcal{C}}^{\square}$ with LEM.

## 2.2 $\mathrm{TT}_{\mathcal{C}}^{\square}$'s Core Syntax and Operational Semantics

$\mathrm{TT}_{\mathcal{C}}^{\square}$'s core syntax and operational semantics are presented in Fig. 1, which for presentation purposes also includes the additional components introduced in Sec. 3, highlighted in blue boxes. Fig. 1's upper part presents the syntax of $\mathrm{TT}_{\mathcal{C}}^{\square}$'s core computation system, where $x$ belongs to a set of variables $\mathsf{Var}$. For simplicity, numbers are considered to be primitive. The constant $\star$ is there for convenience, and is used in place of a term, when the particular term used is irrelevant. Terms are evaluated according to the operational semantics presented in Fig. 1's lower part. In what follows, we use all letters as metavariables for terms. Let $t[x \backslash u]$ stand for the capture-avoiding substitution of all the free occurrences of $x$ in $t$ by $u$.

Types are syntactic forms that are given semantics in Sec. 4 via a forcing interpretation. The type system contains standard types such as dependent products of the form $\mathbf{\Pi} x{:}t_1.t_2$ and dependent sums of the form $\mathbf{\Sigma} x{:}t_1.t_2$. For convenience we write $t_1 \to t_2$ for the non-dependent $\mathbf{\Pi}$ type; $\mathtt{True}$ for $\underline{0}{=}\underline{0}{\in}\mathsf{Nat}$; $\mathtt{False}$ for $\underline{0}{=}\underline{1}{\in}\mathsf{Nat}$; $\neg T$ for $(T \to \mathtt{False})$; $\mathsf{Bool}$ for $\mathtt{True}{+}\mathtt{True}$; $\mathsf{tt}$ for $\mathtt{inl}(\star)$; $\mathsf{ff}$ for $\mathtt{inr}(\star)$; and $\uparrow(t)$ for $t{=}\mathsf{tt}{\in}\mathsf{Bool}$ (a $\mathsf{Bool}$ to type coercion).

Our computation system includes a *space-squashing* mechanism, which we use (among other things) to validate some of the axioms in Secs. 5.1 and 5.2. It erases the evidence that a type is inhabited by truncating it to a subsingleton type using set types: $\downarrow T \coloneqq \{x : \mathtt{True} \mid T\}$. While $\mathtt{True}$ is a contractible type (because equality types are subsingleton types – see Sec. 4), $\downarrow T$ is either empty or inhabited by all (closed) terms in $\mathsf{Term}$, and all its inhabitants are equal to each other. Therefore, $\downarrow T$ is inhabited iff $T$ is inhabited.

Fig. 1's lower part presents $\mathrm{TT}_{\mathcal{C}}^{\square}$'s core small-step operational semantics, where $t_1 \mapsto t_2$ expresses that the term $t_1$ reduces to $t_2$ in one computation step. We omit the congruence rules that allow computing within terms such as: if $t_1 \mapsto t_2$ then $t_1(u) \mapsto t_2(u)$. We denote by $\Downarrow$ the reflexive transitive closure of $\mapsto$, i.e., $a \Downarrow b$ states that $a$ computes to $b$ in $\geq 0$ steps.

# 3 $\mathrm{TT}_{\mathcal{C}}^{\square}$'s Time-Progressing Choice Operators

In addition to the core described in Sec. 2.2, $\mathrm{TT}_{\mathcal{C}}^{\square}$ includes time-progressing notions which we now describe. We capture these notions via the concept of worlds (Sec. 3.1). Then, we

provide a formal, abstract definition of choice operators and add corresponding components to the core system (Sec. 3.2). These time-progressing choice operators cover standard operators such as Brouwerian choice sequences or references (Sec. 3.2.1). We further enrich our system with a notion of time-truncation, used to capture time-sensitive expressions (Sec. 3.3).

## 3.1 Worlds

To capture the time progression notion, the core computation system presented in Sec. 2.2 is parameterized by a Kripke frame [28, 29] defined as follows:

▶ **Definition 1** (Kripke Frame). *A Kripke frame consists of a set of* worlds $\mathcal{W}$ *equipped with a reflexive and transitive binary relation* $\sqsubseteq$.

Let $w$ range over $\mathcal{W}$. We sometimes write $w' \sqsupseteq w$ for $w \sqsubseteq w'$. Let $\mathcal{P}_w$ be the collection of predicates on world extensions, i.e., functions in $\forall w' \sqsupseteq w.\mathbb{P}$. Note that due to $\sqsubseteq$'s transitivity, if $P \in \mathcal{P}_w$ then for every $w' \sqsupseteq w$ it naturally extends to a predicate in $\mathcal{P}_{w'}$. We further define the following notations for quantifiers. $\forall^{\sqsubseteq}_{w}(P)$ states that $P \in \mathcal{P}_w$ is true for all extensions of $w$, i.e., $P\ w'$ holds in all worlds $w' \sqsupseteq w$. $\exists^{\sqsubseteq}_{w}(P)$ states that $P \in \mathcal{P}_w$ is true at an extension of $w$, i.e., $P\ w'$ holds for some world $w' \sqsupseteq w$. For readability, we sometime write $\forall^{\sqsubseteq}_{w}(w'.P)$ (or $\exists^{\sqsubseteq}_{w}(w'.P)$) instead of $\forall^{\sqsubseteq}_{w}(\lambda w'.P)$ (or $\exists^{\sqsubseteq}_{w}(\lambda w'.P)$), respectively.

The operational semantics is parameterized by a frame in the sense that the relation $t_1 \mapsto t_2$ is generalized to a ternary relation between two terms and a world, $t_1 \mapsto_w t_2$, which expresses that $t_1$ reduces to $t_2$ in one step of computation *w.r.t. the world* $w$. Similarly, $a \Downarrow_w b$ generalizes $a \Downarrow b$. We also write $a \Downarrow\!\!\Downarrow_w b$ if $a$ computes to $b$ in all extensions of $w$, i.e., if $\forall^{\sqsubseteq}_{w}(w'.a \Downarrow_{w'} b)$. We write $\sim_w$ for the symmetric and transitive closure of $\Downarrow\!\!\Downarrow_w$.

## 3.2 Time-Progressing Choice Operators

This section introduces the general notion of time-progressing choices into our system. We rely on worlds to record choices and provide operators to access the choices stored in a world. Choices are referred to through their names. A concrete example of such choices are reference cells in programming languages, where a variable name pointing to a reference cell is the name of the corresponding reference cell. To introduce an abstract notion of such choice operators, we assume our computation system contains a set $\mathcal{N}$ of *choice names*, that is equipped with a decidable equality, and an operator that given a list of names, returns a name not in the list. This can be given by, e.g., nominal sets [37]. In what follows we let $\delta$ range over $\mathcal{N}$, and take $\mathcal{N}$ to be $\mathbb{N}$ for simplicity. We introduce further abstract operators and properties in Defs. 2, 4, 8, 10–12, 14, 15, and 19 which our framework is parameterized over, and which we show how to instantiate in Exs. 5, 6, 13, 27, 28, and 30 below. Definitions such as Def. 2 provide axiomatizations of operators, and in addition informally indicate their intended use. Choices are defined abstractly as follows:

▶ **Definition 2** (Choices). *Let* $\mathcal{C} \subseteq \mathsf{Term}$ *be a set of* choices,[1] *and let* $\kappa$ *range over* $\mathcal{C}$. *We say that a computation system contains* $\langle\mathcal{N},\mathcal{C}\rangle$*-choices if there exists a partial function* $\mathsf{choice?} \in \mathcal{W} \to \mathcal{N} \to \mathbb{N} \to \mathcal{C}$. *Given* $w \in \mathcal{W}$, $\delta \in \mathcal{N}$, $n \in \mathbb{N}$, *the returned choice, if it exists, is meant to be the* $n^{th}$ *choice made for* $\delta$ *according to* $w$. $\mathcal{C}$ *is said to be* non-trivial *if it contains two values* $\boldsymbol{\kappa_0}$ *and* $\boldsymbol{\kappa_1}$, *which are computationally different, i.e., such that* $\neg(\boldsymbol{\kappa_0} \sim_w \boldsymbol{\kappa_1})$ *for all* $w$.

[1] To guarantee that $\mathcal{C} \subseteq \mathsf{Term}$, one can for example extend the syntax to include a designated constructor for choices, or require a coercion $\mathcal{C} \to \mathsf{Term}$. We opted for the latter in our formalization.

Thus, to introduce choices into the computation system, we extend the core computation system with a new kind of value for a choice name $\delta$ (as shown in Fig. 1) that can be used to access choices from a world. To facilitate making use of choices extracted from worlds and computing with them, the operational semantics is also extended with the following clause: $\delta(\underline{n}) \mapsto_w \mathsf{choice?}(w, \delta, n)$ (as shown in Fig. 1). This allows applying a choice name $\delta$ to a number $\underline{n}$ to get a choice from the current world $w$. Note that the $\mathbb{N}$ component in this definition enables providing a general notion of choice operators. In some cases, e.g. the case for free-choice sequences, the history is recorded and so the notion of an $n$'s choice is extracted from the history of the choice element. In simpler choice concepts, e.g. references, one only maintains the latest update and so the $\mathbb{N}$ component becomes moot.[2]

We next introduce the notion of a *restriction*, which allows assuming that the choices made for a given choice name all satisfy a pre-defined constraint.

▶ **Definition 3** (Restrictions). *A restriction $r \in \mathsf{Res}$ is a pair $\langle res, d\rangle$ consisting of a function $res \in \mathbb{N} \to \mathcal{C} \to \mathbb{P}$ and a default choice $d \in \mathcal{C}$, such that $\forall(n : \mathbb{N}).(res\ n\ d)$ holds. Given such a pair $r$, we write $r._{\mathsf{d}}$ for $d$; $(r\ n\ \kappa)$ for $(res\ n\ \kappa)$; and $r(\kappa)$ for $\forall(n : \mathbb{N}).r\ n\ \kappa$.*

Intuitively, *res* specifies a restriction on the choices that can be made at any point in time and $d$ provides a default choice that meets this restriction (e.g., for reference cells, this default choice is used to initialize a cell). For example, the restriction $\langle \lambda n.\lambda\kappa.\kappa \in \mathbb{N}, 0\rangle$ requires choices to be numbers and provides 0 as a default value. To reason about restrictions, we require the existence of a "compatibility" predicate as follows.

▶ **Definition 4.** *We further assume the existence of a predicate $\mathsf{compatible} \in \mathcal{N} \to \mathcal{W} \to \mathsf{Res} \to \mathbb{P}$, intended to guarantee that restrictions are satisfied, and which is preserved by $\sqsubseteq$: $\forall(\delta : \mathcal{N})(w_1, w_2 : \mathcal{W})(r : \mathsf{Res}).w_1 \sqsubseteq w_2 \to \mathsf{compatible}(\delta, w_1, r) \to \mathsf{compatible}(\delta, w_2, r)$.*

### 3.2.1 Standard Examples of Choice Operators

The abstract notion of choice operators has many concrete instances. This section provides a high-level description of two such instances: a theoretically-oriented one, based on the notion of free-choice sequences, and a programming-oriented one, based on mutable references.

▶ **Example 5** (Free-Choice Sequences). Free choices are fundamental objects introduced by Brouwer [9] that lay at the heart of intuitionistic mathematics. They are there described as *"new mathematical entities... in the form of infinitely proceeding sequences, whose terms are chosen more or less freely from mathematical entities previously acquired"*. Thus, free-choice sequences are never-finished sequences of objects created *over time* by continuously picking elements from a previously well-defined collection, e.g., the natural numbers. Even though free-choice sequences are ever proceeding, at any point in time the sequence of choices made so far is finite. Therefore, the current state of a choice sequence can be implemented as a list of choices. We use worlds to capture the state of all the choice sequences started so far, and the $\sqsubseteq$ relation on worlds captures the fact that an extension of a world can contain additional choices. In that respect, a choice sequence can be seen as a reference cell that maintains the complete history of values that were stored in the cell. Formally, we define choice sequences of terms, Fcs, as follows (see worldInstanceCS.lagda for details):

[2] Technically, this can be captured by instantiating $\mathcal{C}$ with a function type from $\mathbb{N}$ when records are kept. For simplicity, we here opt to make $\mathbb{N}$ explicit.

**Non-Trivial Choices.** Let $\mathcal{N} \coloneqq \mathbb{N}$ and $\mathcal{C} \coloneqq \mathsf{Term}$, which is non-trivial, e.g., take $\boldsymbol{\kappa_0} \coloneqq \underline{0}$ and $\boldsymbol{\kappa_1} \coloneqq \underline{1}$. Other examples of $\mathcal{C}$s that would be suitable for the results presented in this paper are $\mathbb{N}$, with $\boldsymbol{\kappa_0} \coloneqq 0$ and $\boldsymbol{\kappa_1} \coloneqq 1$ (which can be mapped to the terms $\underline{0}$ and $\underline{1}$); or $\mathbb{B}$ with $\boldsymbol{\kappa_0} \coloneqq \mathsf{true}$ and $\boldsymbol{\kappa_1} \coloneqq \mathsf{false}$ (which can be mapped to the terms $\mathtt{tt}$ and $\mathtt{ff}$).

**Worlds.** Worlds are instantiated as lists of entries, where an entry is either (1) a pair of a choice name and a restriction, indicating the creation of a choice sequence; or (2) a pair of a choice name $\delta$ and a choice $\kappa$ indicating the extension of the choice sequence $\delta$ with the new choice $\kappa$. $\sqsubseteq$ is the reflexive transitive closure of these extension operations. Given an entry list $w$ and a name $\delta$, the state of the choice sequence $\delta$ in $w$ is then the list of extensions made to $\delta$ starting from the point $\delta$ was created in $w$, which allows us to define choice? by looking up the $n^{th}$ choice in that list. This enables starting multiple choice sequences in parallel, which is crucial in the proof of Lem. 16.

**Compatibility.** $\mathsf{compatible}(\delta, w, r)$ states that a choice sequence named $\delta$ with restriction $r$ was started in the world $w$ (using the first kind of entry described above), and that all the choices made for $\delta$ in $w$ satisfy $r$.

▶ **Example 6** (References). Reference cells, which are values that allow a program to indirectly access a particular object, are also choice operators since they can be pointed to different objects over their lifetime. As opposed to a choice sequence, with a reference cell, the history of previous choices is not kept, and the old recorded value is discarded when a new value is stored in a reference cell. In this paper, we will make use of a particular class of reference cell, that are mutable, but can be made immutable at any given point, i.e., the reference cell can be "frozen" so that new values cannot be stored anymore. Formally, we define references to terms, Ref, as follows (see worldInstanceRef.lagda for details):

**Non-trivial Choices.** $\mathcal{N}$ and $\mathcal{C}$ are defined as for free-choice sequences.

**Worlds.** Worlds are lists of cells, where a cell is a quadruple of (1) a choice name, (2) a restriction, (3) a choice, and (4) a Boolean indicating whether the cell is mutable. $\sqsubseteq$ is the reflexive transitive closure of two operations that allow (1) creating a new reference cell, and (2) updating an existing reference cell. We define $\mathsf{choice?}(w, \delta, n)$ so that it simply accesses the content of the $\delta$ cell in $w$, irrespective of what $n$ is. Again, this allows for maintaining multiple reference cells, which is crucial in the proof of Lem. 16.

**Compatibility.** $\mathsf{compatible}(\delta, w, r)$ states that a reference cell named $\delta$ with restriction $r$ was created in the world $w$ (using the first kind of operation described above), and that the current value of the cell satisfies $r$.

## 3.3 Time-Truncation

While some computations are *time-invariant*, in the sense that they compute to the same value at any point in time, others, such as references, are *time-sensitive*. These two kinds of computations have different properties, e.g., a time-invariant term $t$ that computes to a number $\underline{n}$ in a world $w$, will compute to $\underline{n}$ in all $w' \sqsupseteq w$. However, if $t$ is a time-sensitive number, $t$ might compute to numbers different from $\underline{n}$ in extensions of $w$, e.g., $\underline{n{+}1}$ in $w' \sqsupseteq w$ and $\underline{n{+}2}$ in $w'' \sqsupseteq w'$. To capture this distinction at the level of types, we further enrich $\mathrm{TT}^{\square}_{\mathcal{C}}$ by a time-truncation operator $\wr$. The type $\wr T$ contains $T$'s members as well as the terms that behave like members of $T$ at a particular point in time, i.e., in a particular world.

In this paper, we make use in particular of the type $\wr\mathsf{Nat}$, which as opposed to $\mathsf{Nat}$, is not required to only be inhabited by time-invariant terms, and allows for terms to compute to different numbers in different world extensions. For example, $\wr\mathsf{Nat}$ is allowed to be inhabited by a term $t$ that computes to $\underline{3}$ in some world $w$, and to $\underline{4}$ in $w' \sqsupseteq w$. A reference cell

that holds numbers is then essentially of type $\lightning$Nat but not of type Nat, as its content can change over time. This distinction between Nat and $\lightning$Nat will be critical when validating the negation of classical axioms in Sec. 5.1, where we make use of time-sensitive references (in particular in Ex. 13). Note that as we only need a type with two different inhabitants, we could have equally used $\lightning$Bool, whose inhabitants compute to either tt or ff in a given world, but might compute to different Booleans in different extensions.

## 4 The Modality-based Forcing Interpretation

Now that we have defined $TT_{\mathcal{C}}^{\square}$'s computation system that includes choice operators, we provide a semantic for it. $TT_{\mathcal{C}}^{\square}$ is interpreted via a forcing interpretation in which the forcing conditions are worlds. This interpretation is defined using induction-recursion as follows: (1) the inductive relation $w \vDash T_1 \equiv T_2$ expresses type equality in the world $w$; (2) the recursive function $w \vDash t_1 \equiv t_2 \in T$ expresses equality in a type. We further use the following abstractions: $w \vDash \texttt{type}(T)$ for $w \vDash T \equiv T$, $w \vDash t \in T$ for $w \vDash t \equiv t \in T$, and $w \vDash T$ for $\exists(t : \mathsf{Term}).w \vDash t \in T$.

This forcing interpretation is parameterized by a family of abstract modalities $\square$, which we sometimes refer to simply as a modality, which is a function that takes a world $w$ to its modality $\square_w \in \mathcal{P}_w \to \mathbb{P}$. We often write $\square_w(w'.P)$ for $\square_w \lambda w'.P$. To guarantee that this interpretation yields a standard type system in the sense of Thm. 9, we require in Def. 8. that the modalities satisfy certain properties reminiscent of standard modal axiom schemata [16].

The inductive relation $w \vDash T_1 \equiv T_2$ has one constructor per type plus one additional constructor expressing when two types are equal in a world $w$ using the $\square_w$ modality. Consequently, the recursive function $w \vDash t_1 \equiv t_2 \in T$ has as many cases as there are constructors for $w \vDash T \equiv T'$, requiring a dependent version $\boxdot_w^i$ of $\square_w$ to recurse over $i$, which is a proof that $T$ is given meaning using the $\square_w$ modality. Indeed, technically, $\square$ induces two abstract modalities for a world $w$: the modality $\square_w \in \mathcal{P}_w \to \mathbb{P}$, and a dependent version $\boxdot_w^i$, where $P \in \mathcal{P}_w \to \mathbb{P}$ and $i \in \square_w P$. However, to avoid the technical details involved with the dependent modality $\boxdot_w^i$, we opt here for a slightly informal presentation where we slid the technical details concerning the dependent modality to Appx. B.

▶ **Definition 7** (Forcing interpretation). *Given modality $\square$, the forcing interpretation of $TT_{\mathcal{C}}^{\square}$ is given in Fig. 2. There, we write $R^+$ for $R$'s transitive closure, and $\mathsf{Fam}_w(A_1, A_2, B_1, B_2)$ for $w \vDash A_1 \equiv A_2 \wedge \forall_w^{\sqsubseteq}(w'.\forall(a_1, a_2 : \texttt{Term}).w' \vDash a_1 \equiv a_2 \in A_1 \to w' \vDash B_1[x \backslash a_1] \equiv B_2[x \backslash a_2])$.*[3]

There are some standard properties expected for a semantics such as this forcing interpretation to constitute a type system [2, 15]. These include the monotonocity and locality properties expected for a possible-world semantics [44, 18, 17](here monotonicity refers to types, and not to computations). In order to obtain a type system satisfying such standard, useful properties, we must impose some conditions on the modality. Thus, we next identify a set of conditions for the underlying modality that is sufficient for proving these type system properties.

▶ **Definition 8** (Equality modality). *The modality $\square$ is called an* equality modality *if it satisfies the following properties:*

[3] For readability, we adopt a slightly different presentation here compared to the Agda formalization. See Appx. B for a faithful presentation, which in addition covers universes.

[leftmargin=*]

**Numbers:** $w \vDash \text{Nat} \equiv \text{Nat} \iff \texttt{True}$

- $w \vDash t \equiv t' \in \text{Nat} \iff \Box_w(w'.\exists(n:\mathbb{N}).t \Downarrow_{w'} \underline{n} \land t' \Downarrow_{w'} \underline{n})$

**Products:**

- $w \vDash \Pi x{:}A_1.B_1 \equiv \Pi x{:}A_2.B_2 \iff \mathsf{Fam}_w(A_1, A_2, B_1, B_2)$
- $w \vDash f \equiv g \in \Pi x{:}A.B \iff \Box_w(w'.\forall(a_1, a_2 : \mathsf{Term}).w' \vDash a_1 \equiv a_2 \in A \to w' \vDash f\ a_1 \equiv g\ a_2 \in B[x \backslash a_1])$

**Sums:**

- $w \vDash \Sigma x{:}A_1.B_1 \equiv \Sigma x{:}A_2.B_2 \iff \mathsf{Fam}_w(A_1, A_2, B_1, B_2)$
- $w \vDash p_1 \equiv p_2 \in \Sigma x{:}A.B \iff \Box_w(w'.\exists(a_1, a_2, b_1, b_2 : \mathsf{Term}).w' \vDash a_1 \equiv a_2 \in A \land w' \vDash b_1 \equiv b_2 \in B[x \backslash a_1] \land p_1 \Downarrow_{w'} \langle a_1, b_1 \rangle \land p_2 \Downarrow_{w'} \langle a_2, b_2 \rangle)$

**Sets:**

- $w \vDash \{x : A_1 \mid B_1\} \equiv \{x : A_2 \mid B_2\} \iff \mathsf{Fam}_w(A_1, A_2, B_1, B_2)$
- $w \vDash a_1 \equiv a_2 \in \{x : A \mid B\} \iff \Box_w(w'.\exists(b_1, b_2 : \mathtt{Term}).w' \vDash a_1 \equiv a_2 \in A \land w' \vDash b_1 \equiv b_2 \in B[x \backslash a_1])$

**Disjoint unions:**

- $w \vDash A_1 + B_1 \equiv A_2 + B_2 \iff w \vDash A_1 \equiv A_2 \land w \vDash B_1 \equiv B_2$
- $w \vDash a_1 \equiv a_2 \in A + B \iff \Box_w(w'.\exists(u, v : \mathsf{Term}).(a_1 \Downarrow_{w'} \mathtt{inl}(u) \land a_2 \Downarrow_{w'} \mathtt{inl}(v) \land w' \vDash u \equiv v \in A) \lor (a_1 \Downarrow_{w'} \mathtt{inr}(u) \land a_2 \Downarrow_{w'} \mathtt{inr}(v) \land w' \vDash u \equiv v \in B))$

**Equalities:**

- $w \vDash (a_1 = b_1 \in A) \equiv (a_2 = b_2 \in B) \iff w \vDash A \equiv B \land \forall_w^{\sqsubseteq}(w'.w' \vDash a_1 \equiv a_2 \in A) \land \forall_w^{\sqsubseteq}(w'.w' \vDash b_1 \equiv b_2 \in B)$
- $w \vDash a_1 \equiv a_2 \in (a = b \in A) \iff \Box_w(w'.w' \vDash a \equiv b \in A)$ *(note that $a_1$ and $a_2$ can be any term here)*

**Time-Quotiented types:**

- $w \vDash \wr A \equiv \wr B \iff w \vDash A \equiv B$
- $w \vDash a \equiv b \in \wr A \iff \Box_w(w'.(\lambda a, b.\exists(c, d : \mathtt{Value}).a \sim_w c \land b \sim_w d \land w \vDash c \equiv d \in A)^+\ a\ b)$

**Modality closure:**

- $w \vDash T_1 \equiv T_2 \iff \Box_w(w'.\exists(T_1', T_2' : \mathsf{Term}).T_1 \Downarrow_{w'} T_1' \land T_2 \Downarrow_{w'} T_2' \land w' \vDash T_1' \equiv T_2')$
- $w \vDash t_1 \equiv t_2 \in T \iff \Box_w(w'.\exists(T' : \mathtt{Term}).T \Downarrow_{w'} T' \land w' \vDash t_1 \equiv t_2 \in T')$

**Figure 2** Forcing Interpretation.

- $\Box_1$ *(monotonicity of $\Box$):* $\forall(w : \mathcal{W})(P : \mathcal{P}_w).\forall w' \sqsupseteq w. \Box_w P \to \Box_{w'} P$.
- $\Box_2$ *(K, distribution axiom):* $\forall(w : \mathcal{W})(P, Q : \mathcal{P}_w).\Box_w(w'.P\ w' \to Q\ w') \to \Box_w P \to \Box_w Q$
- $\Box_3$ *(C4, i.e., $\Box$ follows from $\Box\Box$):* $\forall(w : \mathcal{W})(P : \mathcal{P}_w).\ \Box_w(w'.\ \Box_{w'} P) \to \Box_w P$
- $\Box_4$: $\forall(w : \mathcal{W})(P : \mathcal{P}_w).\forall_w^{\sqsubseteq}(P) \to \Box_w P$
- $\Box_5$ *(T, reflexivity axiom):* $\forall(w : \mathcal{W})(P : \mathbb{P}).\ \Box_w(w'.P) \to P$

*As detailed in Appx. B, we further require that the dependent modality $\boxdot$ satisfies similar properties to the ones listed above, as well as properties relating the two modalities.*

▶ **Theorem 9.** *Given a computation system with choices $\mathcal{C}$ and an equality modality $\Box$, $TT_{\mathcal{C}}^{\Box}$ is a standard type system in the sense that its forcing interpretation induced by $\Box$ satisfy the*

*following properties (where free variables are universally quantified):*

| | | |
|---|---|---|
| *transitivity:* | $w \vDash T_1 \equiv T_2 \to w \vDash T_2 \equiv T_3 \to w \vDash T_1 \equiv T_3$ | $w \vDash t_1 \equiv t_2 \in T \to w \vDash t_2 \equiv t_3 \in T \to w \vDash t_1 \equiv t_3 \in T$ |
| *symmetry:* | $w \vDash T_1 \equiv T_2 \to w \vDash T_2 \equiv T_1$ | $w \vDash t_1 \equiv t_2 \in T \to w \vDash t_2 \equiv t_1 \in T$ |
| *computation:* | $w \vDash T \equiv T \to T \Downarrow_w T' \to w \vDash T \equiv T'$ | $w \vDash t \equiv t \in T \to t \Downarrow_w t' \to w \vDash t \equiv t' \in T$ |
| *monotonicity:* | $w \vDash T_1 \equiv T_2 \to w \sqsubseteq w' \to w' \vDash T_1 \equiv T_2$ | $w \vDash t_1 \equiv t_2 \in T \to w \sqsubseteq w' \to w' \vDash t_1 \equiv t_2 \in T$ |
| *locality:* | $\Box_w(w'.w' \vDash T_1 \equiv T_2) \to w \vDash T_1 \equiv T_2$ | $\Box_w(w'.w' \vDash t_1 \equiv t_2 \in T) \to w \vDash t_1 \equiv t_2 \in T$ |
| *consistency:* | $\neg w \vDash t \in \texttt{False}$ | |

**Proof.** The proof relies on the properties of the equality modality. For example: $\Box_{\mathbf{1}}$ is used to prove monotonicity when $w \vDash T_1 \equiv T_2$ is derived by closing under $\Box_w$; $\Box_{\mathbf{2}}$ and $\Box_{\mathbf{4}}$ are used, e.g., to prove the symmetry and transitivity of $w \vDash t \equiv t' \in \mathsf{Nat}$; $\Box_{\mathbf{3}}$ is used to prove locality; and $\Box_{\mathbf{5}}$ is used to prove consistency. See props3.lagda for further details. ◀

# 5 Compatibility with Classical Axioms

To study the compatibility of $\mathrm{TT}_{\mathcal{C}}^{\Box}$ with classical reasoning, this section identifies two subclasses of the family of type theories $\mathrm{TT}_{\mathcal{C}}^{\Box}$, specified through conditions on the choices and modalities. Sec. 5.1 provides conditions that are sufficient to derive the negation of classical axioms such as LEM, while Sec. 5.2 provides conditions that are sufficient to derive LEM. We further give concrete instantiations for such choices and modalities (the modalities are instantiated only in Sec. 6.2 based on the notion of bars).

## 5.1 Intuitionistic Theories

This section identifies a set of general properties of choices and modalities that enables proving the negation of classical axioms such as LEM. We call theories based on such choices and modalities "intuitionistic", in the sense that they are incompatible with classical reasoning.

The proof of the negation of classical axioms provided below (Cor. 17) captures intuitionistic counterexamples [22, 8] abstractly. Briefly, we prove that, given a non-trivial choice structure, (A) if the only choice made so far is $\boldsymbol{\kappa_0}$, then it is not possible to decide whether $\boldsymbol{\kappa_1}$ will ever be made. More precisely, we prove that: (B) it is not the case that $\boldsymbol{\kappa_1}$ will be made because there are extensions where it won't; and (C) it is not the case that $\boldsymbol{\kappa_1}$ is not made in all extensions because there are extensions where it is made. To capture this, we require some additional properties from the underlying choices and modalities. To ensure that (A) holds, we introduce an *extendability* property in Def. 10, which allows creating a fresh choice name $\delta$ and a world $w$ where the only choice made for $\delta$ in $w$ is $\boldsymbol{\kappa_0}$. (B) is proved thanks to the properties introduced in Defs. 14 and 15, which guarantee the existence of an extension where the $n^{th}$ choice made for $\delta$ is $\boldsymbol{\kappa_0}$, for any $n \in \mathbb{N}$. (C) is proved using the *immutability* property in Def. 11, which allows exhibiting a world where $\boldsymbol{\kappa_1}$ is made.

▶ **Definition 10** (Extendability). *We say that $C$ is* extendable *if there exists a function $\nu\mathcal{C} \in \mathcal{W} \to \mathcal{N}$, where $\nu\mathcal{C}(w)$ is intended to return a new choice name not present in $w$, and a function $\mathsf{start}\nu\mathcal{C} \in \mathcal{W} \to \mathsf{Res} \to \mathcal{W}$, where $\mathsf{start}\nu\mathcal{C}(w,r)$ is intended to return an extension of $w$ with the new choice name $\nu\mathcal{C}(w)$ with restriction $r$, satisfying the following properties:*

- *Starting a new choice extends the current world:* $\forall(w:\mathcal{W})(r:\mathsf{Res}).w \sqsubseteq \mathsf{start}\nu\mathcal{C}(w,r)$
- *Initially, the only possible choice is the default value of the given restriction, i.e.:* $\forall(n:\mathbb{N})(r:\mathsf{Res})(w:\mathcal{W})(\kappa:\mathcal{C}).\mathsf{choice?}(\mathsf{start}\nu\mathcal{C}(w,r),\nu\mathcal{C}(w),n) = \kappa \to \kappa = r_{.\mathsf{d}}$
- *A choice is initially compatible with its restriction:* $\forall(w:\mathcal{W})(r:\mathsf{Res}).\mathsf{compatible}(\nu\mathcal{C}(w),\mathsf{start}\nu\mathcal{C}(w,r),r)$

If only one choice $\kappa$ was made so far for a name $\delta$, then to prove (C) above we exhibit an extension where another choice $\kappa'$ is made. Thus, we require a way to make a choice $\kappa' \neq \kappa$, as well as a way to make $\kappa'$ immutable in the sense that no other choice than $\kappa'$ can be made in the future. This is necessary because $\mathrm{TT}_{\mathcal{C}}^{\square}$ is a monotonic theory (see Lem. 16's proof). Consequently, we further rely on the ability to, at any point in time, be able to constrain the choices to be the same forever. This does not prevent making different choice before a choice is made immutable, and the ability to make different choices over time is indeed necessary as we just highlighted. To capture this, we define the immutability property.

▶ **Definition 11** (Immutability). *We say that $C$ is immutable if there exist a function $\mathsf{freeze} \in \mathcal{N} \to \mathcal{C} \to \mathcal{W} \to \mathcal{W}$ (where $\mathsf{freeze}(\delta, \kappa, w)$ is intended to return a world $w'$ that extends the world $w$ with the choice $\kappa$ for the choice name $\delta$, and such that $\kappa$ can be retrieved in any extension of $w'$), and a predicate $\mathsf{mutable} \in \mathcal{N} \to \mathcal{W} \to \mathbb{P}$ (intended to hold iff the choice name is mutable in the world, i.e., different choices can be made), satisfying the following properties:*

- *Making an immutable choice extends the current world:* $\forall(\delta : \mathcal{N})(w : \mathcal{W})(\kappa : \mathcal{C})(r : \mathsf{Res}).\mathsf{compatible}(\delta, w, r) \to r(\kappa) \to w \sqsubseteq \mathsf{freeze}(\delta, \kappa, w)$
- *A choice is initially mutable:* $\forall(w : \mathcal{W})(r : \mathsf{Res}).\mathsf{mutable}(\nu\mathcal{C}(w), \mathsf{start}\nu\mathcal{C}(w, r))$
- *Immutable choices stay immutable:* $\forall(\delta : \mathcal{N})(w : \mathcal{W})(\kappa : \mathcal{C})(r : \mathsf{Res}).\mathsf{compatible}(\delta, w, r) \to \mathsf{mutable}(\delta, w) \to \exists(n : \mathbb{N}).\forall^{\sqsubseteq}_{\mathsf{freeze}(\delta,\kappa,w)}(w'.\mathsf{choice?}(w', \delta, n) = \kappa)$

In addition, to state properties about non-trivial choices within $\mathrm{TT}_{\mathcal{C}}^{\square}$, such as the fact that it is not always decidable whether a choice will be made in the future (see $\Sigma\mathsf{choice}$ in Lem. 16), we assume the existence of a term ($\in \mathsf{Term}$) denoting a type that contains the two distinct choices $\boldsymbol{\kappa_0}$ and $\boldsymbol{\kappa_1}$, capturing Def. 2 at the level of the theory $\mathrm{TT}_{\mathcal{C}}^{\square}$.

▶ **Definition 12** (Reflection). *We say that $C$ is reflected if there exists a term $\mathsf{Type}\mathcal{C} \in \mathsf{Term}$ such that the following hold for all worlds $w$:*

- *$\mathsf{Type}\mathcal{C}$ is a type inhabited by $\boldsymbol{\kappa_0}$ and $\boldsymbol{\kappa_1}$:* $w \vDash \mathtt{type}(\mathsf{Type}\mathcal{C})$, $w \vDash \boldsymbol{\kappa_0} \in \mathsf{Type}\mathcal{C}$, $w \vDash \boldsymbol{\kappa_1} \in \mathsf{Type}\mathcal{C}$.
- *The choices that inhabit $\mathsf{Type}\mathcal{C}$ are related w.r.t. $\sim$:* $\forall(w : \mathcal{W})(a, b : \mathsf{Term}).w \vDash a{\equiv}b{\in}\mathsf{Type}\mathcal{C} \to \square_w(w'.\forall^{\sqsubseteq}_{w'}(w''.\forall(\kappa_1, \kappa_2 : \mathcal{C}).a \Downarrow_{w''} \kappa_1 \to b \Downarrow_{w''} \kappa_2 \to \kappa_1 \sim_{w''} \kappa_2))$
- *Choices obtained from worlds that compute to either $\boldsymbol{\kappa_0}$ or $\boldsymbol{\kappa_1}$ inhabit $\mathsf{Type}\mathcal{C}$:* $\forall(w : \mathcal{W})(n : \mathbb{N})(\delta : \mathcal{N}).\ \square_w\ (w'.(\mathsf{choice?}(w', \delta, n) \Downarrow_{w'} \boldsymbol{\kappa_0} \vee \mathsf{choice?}(w', \delta, n) \Downarrow_{w'} \boldsymbol{\kappa_1})) \to w \vDash (\delta(\underline{n})){\in}\mathsf{Type}\mathcal{C}$

Crucially, these properties allow $\mathsf{Type}\mathcal{C}$'s inhabitants to be time-sensitive, i.e., to compute to different choices in different extensions, which allows implementing choices with either references or choice sequences. As shown in Ex. 13, we can then instantiate $\mathsf{Type}\mathcal{C}$ with ⇂-truncated types, which references inhabit.

Building up on the examples of choice operators presented in Exs. 5 and 6, we next provide examples for the aforementioned properties of choices.

▶ **Example 13.** Both free-choice sequences, Fcs, and references, Ref, are extendable, immutable and reflected choices.

**Extendable.** $\nu\mathcal{C}(w)$ returns a choice name not occurring in $w$. For Fcs, $\mathsf{start}\nu\mathcal{C}(w, r)$ adds a new entry to $w$ that creates a choice sequence with name $\nu\mathcal{C}(w)$ and restriction $r$ (using the first kind of entry mentioned in Ex. 5). For Ref, $\mathsf{start}\nu\mathcal{C}(w, r)$ adds a new reference cell to $w$ with name $\nu\mathcal{C}(w)$ and restriction $r$ (using the first kind of operation mentioned in Ex. 6). In both cases, the properties are straightforward.

**Immutable.** For Fcs, $\mathsf{freeze}(\delta, \kappa, w)$ extends $w$ with a new entry (of the second kind from Ex. 5) that adds a new choice $\kappa$ to the choice sequence $\delta$. $\mathsf{mutable}(\delta, w)$ is always true since it is always possible to extend choice sequences with new choices. For Ref, $\mathsf{freeze}(\delta, \kappa, w)$ updates $w$ by changing the content of the reference cell $\delta$ to $\kappa$ if it is mutable and marking it as immutable; and $\mathsf{mutable}(\delta, w)$ checks that $\delta$ is still mutable in $w$.

**reflected.** $\mathsf{Type}\mathcal{C}$ is $\wr\mathsf{Nat}$ in both cases, which is inhabited by $\boldsymbol{\kappa_0} \coloneqq \underline{0}$ and $\boldsymbol{\kappa_1} \coloneqq \underline{1}$. The other properties follow from the semantics of $\wr\mathsf{Nat}$. The use of $\wr$ is crucial because without it we would not be able to prove that choices obtained from worlds that compute to either $\boldsymbol{\kappa_0}$ or $\boldsymbol{\kappa_1}$ inhabit $\mathsf{Type}\mathcal{C}$, as reference cells can change value over time.

Next, we define the following two properties, which among other things allow proving (B) above. Sec. 6.2.1 shows how those properties can be proved for concrete instances of $\Box$ with Beth bars. The first property requires that the choices corresponding to a name on which a restriction $r$ is imposed, can always eventually be retrieved and that they satisfy $r$.

▶ **Definition 14** (Retrieving). *The modality $\Box$ is called retrieving if:*
$\forall(w : \mathcal{W})(\delta : \mathcal{N})(n : \mathbb{N})(r : \mathsf{Res}).\mathsf{compatible}(\delta, w, r) \to \Box_w(w'.r\ n\ \mathsf{choice?}(w', \delta, n))$

The second property states that if $\Box_w P$ then $P$ is true in an extension of $w$, and this for a specific class of worlds, namely those where only one choice has been made so far (possibly multiple times) and is still mutable. This property allows following a sequence of worlds where the same choice is picked for a given choice name.

▶ **Definition 15** (Choice-following). *The modality $\Box$ is called choice-following if:*
$\forall(\delta : \mathcal{N})(w : \mathcal{W})(P : \mathcal{P}_w)(r : \mathsf{Res}).\mathsf{Sat}(w, \delta, r) \to \Box_w P \to \exists^{\sqsubseteq}_w(w'.P\ w' \wedge \mathsf{Sat}(w', \delta, r))$
*where* $\mathsf{Sat}(w, \delta, r) \coloneqq \mathsf{compatible}(\delta, w, r) \wedge \mathsf{mutable}(\delta, w) \wedge \mathsf{OnlyChoice}(w, \delta, r_{.\mathsf{d}})$
*and* $\mathsf{OnlyChoice}(w, \delta, \kappa) \coloneqq \forall(n : \mathbb{N})(\kappa' : \mathcal{C}).\mathsf{choice?}(w, \delta, n) = \kappa' \to \kappa' = \kappa$.

Before we prove the negation of classical axioms, we first prove the following general result. Note the use of $\downarrow$ in Lem. 16, where $\downarrow(T{+}U)$ captures a classical reading of "or".

▶ **Lemma 16.** *Let $TT^{\Box}_{\mathcal{C}}$ be a type system where $\mathcal{C}$ is a non-trivial, extendable, immutable and reflected set of choices and $\Box$ is a retrieving, choice-following equality modality. Then, the followings hold (see not_lem.lagda for details):*

- $\forall(w : \mathcal{W}).\neg\ \Box_{\mathsf{start}\nu\mathcal{C}(w,r)}\ (w'.(w' \vDash \Sigma\mathcal{C}(w)) \vee \forall^{\sqsubseteq}_{w'}(w''.\neg w'' \vDash \Sigma\mathcal{C}(w)))$
- $\forall(w : \mathcal{W}).\neg\mathsf{start}\nu\mathcal{C}(r, w) \vDash \downarrow(\Sigma\mathcal{C}(w){+}\neg\Sigma\mathcal{C}(w))$

*where (1)* $\Sigma\mathsf{choice}(\delta, \kappa) \coloneqq \Sigma k{:}\mathsf{Nat}.((\delta(k)){=}\kappa{\in}\mathsf{Type}\mathcal{C})$; *(2)* $\Sigma\mathcal{C}(w) \coloneqq \Sigma\mathsf{choice}(\nu\mathcal{C}(w), \boldsymbol{\kappa_1})$; *and (3)* $r \coloneqq \langle res, d\rangle$ *is the restriction where* $res \coloneqq \lambda n, \kappa.(\kappa = \boldsymbol{\kappa_0} \vee \kappa = \boldsymbol{\kappa_1})$ *and* $d \coloneqq \boldsymbol{\kappa_0}$.

**Proof.** As the second statement is a straightforward consequence of the first, we only sketch a proof of the first. Let $w \in \mathcal{W}$. By extendability, we derive a new choice name $\delta$, namely $\nu\mathcal{C}(w)$, and an extension $\mathsf{start}\nu\mathcal{C}(w, r)$ of $w$, where the only choice made so far for $\delta$ is $\boldsymbol{\kappa_0}$, and such that $\mathsf{mutable}(\delta, \mathsf{start}\nu\mathcal{C}(w, r))$, by immutability. We assume $\Box_{\mathsf{start}\nu\mathcal{C}(w,r)}(w'.(w' \vDash \Sigma\mathcal{C}(w)) \vee \forall^{\sqsubseteq}_{w'}(w''.\neg w'' \vDash \Sigma\mathcal{C}(w)))$, and by the choice-following property we can derive a world $w' \sqsupseteq \mathsf{start}\nu\mathcal{C}(w, r)$, where the only choice made so far for $\delta$ is $\boldsymbol{\kappa_0}$, and such that $w' \vDash \Sigma\mathcal{C}(w)$ or $\forall^{\sqsubseteq}_{w'}(w''.\neg w'' \vDash \Sigma\mathcal{C}(w))$. We now derive a contradiction in both cases:

- $w' \vDash \Sigma\mathcal{C}(w)$: By the choice-following property and the meaning of $\Sigma\mathcal{C}(w)$, we derive that there exists $k \in \mathbb{N}$ such that $\delta(\underline{k})$ and $\boldsymbol{\kappa_1}$ are equal members of the type $\mathsf{Type}\mathcal{C}$ in some world $w'' \sqsupseteq w'$, where the only choice so far associated with $\delta$ is $\boldsymbol{\kappa_0}$. Since the modality is retrieving and choice-following, we can further derive a world $w''' \sqsupseteq w''$ where $\delta(\underline{k})$ computes to a choice $\kappa$ satisfying $r$ (therefore, either $\kappa = \boldsymbol{\kappa_0}$ or $\kappa = \boldsymbol{\kappa_1}$), and again where the only choice so far associated with $\delta$ is $\boldsymbol{\kappa_0}$. We derive that $\delta(\underline{k})$ computes to $\boldsymbol{\kappa_0}$, which cannot be equal to $\boldsymbol{\kappa_1}$, from which we obtain a contradiction.

- $\forall^{\sqsubseteq}_{w'}(w''.\neg w'' \vDash \Sigma\mathcal{C}(w))$: By immutability, we build the world $w'' = \mathsf{freeze}(\delta, \kappa_1, w') \sqsupseteq w'$, and get to assume $\neg w'' \vDash \Sigma\mathsf{choice}(\delta, \kappa_1)$. The reflected choice and retrieving modality entail $w'' \vDash \Sigma\mathsf{choice}(\delta, \kappa_1)$, from which we conclude a contradiction. Let us comment on the use of freeze. Assume that when "freezing" $\kappa_1$, it is the $n^{th}$ choice being made for $\delta$ in $w''$. Then, $(\delta\ \underline{n})$ computes to $\kappa_1$ in $w''$. To derive $w'' \vDash \Sigma\mathsf{choice}(\delta, \kappa_1)$ we must prove that $(\delta\ \underline{n})$ computes to $\kappa_1$, which using $\Box_3$, we must do in a $w''' \sqsupseteq w''$. Now, as some computations are time-sensitive (such as those involving references), without immutability it might not be that $(\delta\ \underline{n})$ computes to $\kappa_1$ in $w'''$. ◀

Using Lem. 16, we can derive the negation of classical axioms such as LEM, or the Limited Principle of Omniscience (LPO) [6, p.9] (the above examples showed how to prove some of the assumptions in this lemma for instances of $\mathcal{C}$ and $\Box$, and the others are described in Sec. 6.2.1, as they rely on a concrete instance of $\Box$ with Beth bars).

▶ **Corollary 17** (Incompatibility with Classical Principles). *Let $TT^{\Box}_{\mathcal{C}}$ be a type system where $\mathcal{C}$ is a non-trivial, extendable, immutable and reflected set of choices and $\Box$ is a retrieving, choice-following modality. Then, the following hold (see not_lem.lagda and not_lpo.lagda):*

- *¬LEM:* $\forall(w:\mathcal{W}).\neg w \vDash \mathbf{\Pi} P{:}\mathbb{U}_i.{\downarrow}(P + \neg P)$
- *¬LPO:* $\forall(w:\mathcal{W}).\neg w \vDash \mathbf{\Pi} f{:}\mathsf{Nat} \to \mathsf{Bool}.{\downarrow}((\mathbf{\Sigma} n{:}\mathsf{Nat}.{\uparrow}(f\ n)) + (\mathbf{\Pi} n{:}\mathsf{Nat}.\neg{\uparrow}(f\ n)))$

*For LPO, we further assume that choices are Booleans, i.e., that Type$\mathcal{C}$ from Def. 12 is* Bool, *that $\kappa_0$ is* tt *and that $\kappa_1$ is* ff *(see Remark 18 for further details).*

▶ Remark 18. As mentioned in Cor. 17, to prove ¬LPO we further assume that choices are Booleans, i.e., Type$\mathcal{C}$ from Def. 12 is Bool, $\kappa_0$ is tt and $\kappa_1$ is ff. This is due to the fact that LPO is stated in terms of a function in Nat → Bool, which we instantiate with a choice sequence whose choices are restricted to Booleans to prove its negation. This is possible because a free choices sequence name $\delta$ occurring in a world with a restriction constraining its choices to be Booleans has type Nat → Bool because choices do not change over time. However, a reference name $\delta$ occurring in world with a restriction constraining its choices to be Booleans has type Nat → $\wr$Bool, and not Nat → Bool, because its choices can change over time. However, we can prove the following alternative version of ¬LPO, where $\Uparrow(T) := T{=}\mathsf{tt}{\in}\wr\mathsf{Bool}$, using references (see not_lpo_qtbool.lagda for details):

$$\forall(w:\mathcal{W}).\neg w \vDash \mathbf{\Pi} f{:}\mathsf{Nat} \to \wr\mathsf{Bool}.{\downarrow}((\mathbf{\Sigma} n{:}\mathsf{Nat}.\Uparrow(f\ n)) + (\mathbf{\Pi} n{:}\mathsf{Nat}.\neg\Uparrow(f\ n)))$$

Furthermore, using results similar to the ones presented in Lem. 16, we can prove the negation of Markov's Principles (see not_mp.lagda for details):

$$\forall(w:\mathcal{W}).\neg w \vDash \mathbf{\Pi} f{:}\mathsf{Nat} \to \mathsf{Bool}.(\neg\mathbf{\Pi} n{:}\mathsf{Nat}.\neg{\uparrow}(f\ n)) \to {\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.{\uparrow}(f\ n)$$

In addition to requiring that choices are Booleans as for LPO, the proof also requires that mutable is always true (even if we had used $\wr$Bool instead of Bool), which only holds about free-choice sequences but not references.

## 5.2 Agnostic Theories

This section introduces the following general property of modalities that enables proving LEM, leading to "agnostic" instances of $\mathrm{TT}^{\Box}_{\mathcal{C}}$, in the sense that they support classical reasoning.

▶ **Definition 19** (Jumping). *The modality $\Box_w$ is called jumping if:*
$\forall(w:\mathcal{W})(P:\mathcal{P}_w).\forall^{\sqsubseteq}_{w}(w_1.\exists^{\sqsubseteq}_{w_1}(w_2.\ \Box_{w_2} P)) \to \Box_w P$

Note that, classically, the negation of the choice-following property can be read as: $\exists(\delta : \mathcal{N})(w : \mathcal{W})(P : \mathcal{P}_w)(r : \mathsf{Res}).\mathsf{Sat}(w,\delta,r) \wedge \Box_w P \wedge \forall^{\sqsubseteq}_w(w'.\mathsf{Sat}(w',\delta,r) \to \neg(P\ w'))$. Reading $\Box$ as "always eventually" this says that there exists a property $P$, which is always eventually true but there is no extension of the current world that satisfies Sat where $P$ is true. Thus, not all possible futures have to be covered for a property to be "always eventually" true. The jumping property captures a similar behavior only requiring to prove that for all $w_1 \sqsupseteq w$ it is enough to exhibit one world $w_2 \sqsupseteq w_1$ where $P$ is "always eventually" true, to derive that $P$ is "always eventually" true. We now prove that $\mathrm{TT}^{\Box}_{\mathcal{C}}$ is compatible with LEM when instantiated with jumping modalities.

▶ **Lemma 20** (Compatibility with LEM). *Let $TT^{\Box}_{\mathcal{C}}$ be a type system where $\Box_w$ is a jumping equality modality. Then, the following holds (classically): $\forall(w : \mathcal{W}).w \vDash \mathbf{\Pi} P{:}\mathbb{U}_i.{\downarrow}(P{+}\neg P)$.*

**Proof.** By the semantics of the $\mathbf{\Pi} P{:}\mathbb{U}_i.{\downarrow}(P{+}\neg P)$, it is enough to prove that for all $w \in \mathcal{W}$ and $p \in \mathsf{Term}$ such that $w \vDash p{\in}\mathbb{U}_i$, then $\Box_w(w'.w' \vDash p \vee \forall^{\sqsubseteq}_{w'}(w''.\neg w'' \vDash p))$. By the jumping property, it is enough to prove $\forall^{\sqsubseteq}_w(w_1.\exists^{\sqsubseteq}_{w_1}(w_2.\Box_{w_2}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p))))$. Let $w_1 \sqsupseteq w$, and we prove $\exists^{\sqsubseteq}_{w_1}(w_2.\Box_{w_2}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p)))$. Using classical logic, we can then prove this by cases (see lem.lagda for further details):

- $\exists^{\sqsubseteq}_{w_1}(w_2.w_2 \vDash p)$: We obtain a $w_2 \sqsupseteq w_1$ such that $w_2 \vDash p$. We instantiate our conclusion using $w_2$, and must prove $\Box_{w_2}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p))$. Using $\Box_{\mathbf{4}}$ it is enough to prove $\forall^{\sqsubseteq}_{w_2}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p))$, which we prove by monotonicity of $w_2 \vDash p$.
- $\neg\exists^{\sqsubseteq}_{w_1}(w_2.w_2 \vDash p)$: We instantiate our conclusion using $w_1$, and show that $\Box_{w_1}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p))$. Using $\Box_{\mathbf{4}}$, it is enough to prove $\forall^{\sqsubseteq}_{w_1}(w_3.w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p))$. Therefore, assuming $w_3 \sqsupseteq w_1$, it remains to show $w_3 \vDash p \vee \forall^{\sqsubseteq}_{w_3}(w_4.\neg w_4 \vDash p)$, and since the right disjunct is provable, this contradicts our assumption. ◀

## 6 Bars

The notion of topological spaces of bars is typically used in possible worlds semantics to capture the intuitive notion of time progression and provide a forcing interpretation. Therefore, this section provides an abstract definition of this notion and establishes the connection to the aforementioned equality modalities. Concretely, we offer a notion of monotone bars that we then use to instantiate the equality modalities with.

### 6.1 Bar Spaces

The opens of a topological bar space are collections of worlds. To define a topological space of bars, one needs to describe the "shape" of the opens in the space through a predicate, which specifies when an open belongs to the space. Given a bar space, a bar in that space is an open (a collection of worlds) that satisfies the predicate specifying the space.

▶ **Definition 21** (Bars). *Let $\mathcal{O} \coloneqq \mathcal{W} \to \mathbb{P}$ be the set of predicates on worlds, which we call opens, and let $\mathsf{BarProp} \coloneqq \mathcal{W} \to \mathcal{O} \to \mathbb{P}$ be the set of predicates on opens. An open $o$ is said to be a* bar *in $B \in \mathsf{BarProp}$ w.r.t. a world $w$ if: (1) it satisfies $(B\ w\ o)$, (2) all its elements extend $w$, and (3) it is upward closed w.r.t. $\sqsubseteq$ (i.e., if $w_1 \sqsubseteq w_2$ and $(o\ w_1)$ then $(o\ w_2)$). We denote the set of all bars in $B$ w.r.t. $w$ by $\mathcal{B}^w_B$.*

Intuitively, given $B \in \mathsf{BarProp}$, $(B\ w\ o)$ specifies whether $o$ "bars" the world $w$. We write $w{\triangleleft}o{\in}B$ for $(B\ w\ o)$, and $w' \in o$ for $(o\ w')$.

▶ **Definition 22** (Bar Spaces). *$B \in \mathsf{BarProp}$ is called a* bar space *if it satisfies the followings:*

- $\mathsf{isect}(B) \coloneqq \forall(w : \mathcal{W})(o_1, o_2 : \mathcal{O}).w \triangleleft o_1 \in B \to w \triangleleft o_2 \in B \to w \triangleleft (o_1 \cap o_2) \in B$, *where* $o_1 \cap o_1 \in \mathcal{O} \coloneqq \lambda w_0.\exists(w_1, w_2 : \mathcal{W}).w_1 \in o_1 \wedge w_2 \in o_2 \wedge w_1 \sqsubseteq w_0 \wedge w_2 \sqsubseteq w_0$.
- $\mathsf{union}(B) \coloneqq \forall(w : \mathcal{W})(b : \mathcal{B}_B^w)(i : \forall w' \sqsupseteq w.w' \in b \to \mathcal{B}_B^{w'}).w \triangleleft (\cup(i)) \in B$, *where* $\cup(i) \in \mathcal{O} \coloneqq \lambda w_0.\exists w_1 \sqsupseteq w.\exists(j : w_1 \in b).w_0 \in (i\ w_1\ j)$, *given* $i \in \forall w' \sqsupseteq w.w' \in b \to \mathcal{B}_B^{w'}$.
- $\mathsf{top}(B) \coloneqq \forall(w : \mathcal{W}).w \triangleleft (\top(w)) \in B$, *where* $\top(w) \in \mathcal{O} \coloneqq \lambda w_0.w \sqsubseteq w_0$.
- $\mathsf{non}\varnothing(B) \coloneqq \forall(w : \mathcal{W})(b : \mathcal{B}_B^w).\exists_w^{\sqsubseteq}(w'.w' \in b)$.
- $\mathsf{sub}(B) \coloneqq \forall(w_1, w_2 : \mathcal{W})(o : \mathcal{O}).w_1 \sqsubseteq w_2 \to w_1 \triangleleft o \in B \to w_2 \triangleleft (o\downharpoonleft_{w_2}) \in B$, *where* $o\downharpoonleft_w \in \mathcal{O} \coloneqq \lambda w_0.\exists(w_1 : \mathcal{W}).w_1 \in o \wedge w_1 \sqsubseteq w_0 \wedge w \sqsubseteq w_0$.

*We denote by* BarSpace *the set of all bar spaces.*

That is, a bar space $B$ is a set of opens that is closed under binary intersections (i.e., $\mathsf{isect}(B)$) and arbitrary unions (i.e., $\mathsf{union}(B)$), contains a top element (i.e., $\mathsf{top}(B)$), all its elements are non-empty (i.e., $\mathsf{non}\varnothing(B)$), and is closed under subsets (i.e., $\mathsf{sub}(B)$).

For $w \in \mathcal{W}$, $P \in \mathcal{P}_w$, $B \in \mathsf{BarSpace}$, and $b \in \mathcal{B}_B^w$, we write $P \in b$ for $\forall w' \sqsupseteq w.w' \in b \to P\ w'$, i.e., $P$ holds at the bar $b$, i.e., for all elements in $b$. Let $\exists\mathcal{B}_B^w \in \mathcal{P}_w \to \mathbb{P}$ be defined as $\lambda P.(\exists(b : \mathcal{B}_B^w).P \in b)$, i.e., that $P$ holds in some bar of the space $B$. Using this definition, we next show that any bar space $B$ induces an equality modality.

▶ **Proposition 23.** *If* $B \in \mathsf{BarSpace}$ *and* $w \in \mathcal{W}$*, then* $\exists\mathcal{B}_B^w$ *is an equality modality.*

**Proof.** Given the properties of a bar space, we derive corresponding properties for bars in $\mathcal{B}_B^w$, and in turn, the properties of an equality modality. In particular, $\mathsf{sub}(B)$ allows deriving $\square_\mathbf{1}$, $\mathsf{isect}(B)$ allows deriving $\square_\mathbf{2}$, $\mathsf{union}(B)$ allows deriving $\square_\mathbf{3}$, $\mathsf{non}\varnothing(B)$ allows deriving $\square_\mathbf{5}$, and $\mathsf{top}(B)$ allows deriving $\square_\mathbf{4}$. See Appx. C and bar.lagda for further details. ◀

Let $\mathrm{TT}_\mathcal{C}^B$ be the theory $\mathrm{TT}_\mathcal{C}^\square$, where $\square$ is derived from $B \in \mathsf{BarSpace}$ using Prop. 23.

▶ **Corollary 24.** *For any choice operator* $\mathcal{C}$ *and* $B \in \mathsf{BarSpace}$*,* $TT_\mathcal{C}^B$ *is a type system in the sense of Thm. 9.*

## 6.2 Examples of Bar Spaces

We next present three bar space examples, namely Beth bars in Def. 26, open bars in Def. 29, and Kripke bars in Def. 31, and use them to provide concrete instances for intuitionistic and agnostic theories. In particular, we show that the choice-following property, which is key in proving compatibility with LEM, is satisfied by Beth bars but not by open bars.

### 6.2.1 Beth Bars

As presented below, a Beth bar is defined so that for any infinite sequence of worlds ordered by $\sqsubseteq$, there exists a world in that sequence belonging to the bar. However, for Beth bars to satisfy the retrieving property presented in Def. 14, we must also ensure that for any choice name $\delta$ occurring in a world $w$ in a chain, there is a $w' \sqsupseteq w$ in that chain such that $\mathsf{choice?}(w', \delta, n)$ is defined. To this end we introduce a predicate $\mathsf{progress} \in \mathcal{N} \to \mathcal{W} \to \mathcal{W} \to \mathbb{P}$, which we show how to instantiate in Exs. 27 and 28, as well as the concept of (progressing) chains:

▶ **Definition 25** (Chains & Barred Chains)**.** *Let* $\mathsf{chain}(w)$ *be the set of sequences of worlds in* $\mathbb{N} \to \mathcal{W}$ *such that* $c \in \mathsf{chain}(w)$ *iff (1)* $w \sqsubseteq c\ 0$*, (2) for all* $i \in \mathbb{N}$*,* $c\ i \sqsubseteq c\ (i+1)$*; and (3)* $c$ *is* progressing*, i.e.,* $\forall(\delta : \mathcal{N})(n : \mathbb{N})(r : \mathsf{Res}).\mathsf{compatible}(\delta, (c\ n), r) \to \exists m > n.\mathsf{progress}(\delta, (c\ n), (c\ m))$*. We say that a chain* $c \in \mathsf{chain}(w)$ *is* barred *by an* $o \in \mathcal{O}$*, denoted* $\mathsf{barredChain}(o, c)$*, if there exists a world* $w' \sqsubseteq (c\ n)$ *for some* $n \in \mathbb{N}$ *such that* $w' \in o$*.*

Using chains, we define Beth bars as follows:

▶ **Definition 26** (Beth Bars). *Beth bars are defined by the following bar predicate* Beth $:=$ $\lambda w.\lambda o.\forall(c : \mathsf{chain}(w)).\mathsf{barredChain}(o, c)$*, which is a bar space due to the properties of chains.*[4]

We now show through the following two examples how to define Beth bars, and how they induce a retrieving (Def. 14) and choice-following (Def. 15) modality, as required by Cor. 17.

▶ **Example 27** (Beth Bars & Free-Choice Sequences). Building up on Ex. 13, we present here an example where choices are free-choice sequences and bars are Beth bars, yielding an intuitionistic theory $\mathrm{TT}^{\mathsf{Beth}}_{\mathsf{Fcs}}$ (see worldInstanceCS.lagda and modInstanceBethCs.lagda for details). This is the theory presented in [4].

**Progress.** For Fcs, $\mathsf{progress}(\delta, w_1, w_2)$ states that the state of the choice sequence $\delta$ in $w_1$ is a strict initial segment of the state of the choice sequence $\delta$ in $w_2$.

**Retrieving.** We prove this property by exhibiting a bar that given a choice name $\delta$ and a $n \in \mathbb{N}$, requires its $n^{th}$ choice to exist. We can prove that this forms a Beth bar thanks to the fact that chains are required to always eventually make progress.

**Choice-following.** This property is true about Beth bars because they require *all* possible chains of worlds extending a given world $w$ to be "barred" by the bar. Given a choice name $\delta$ that satisfies $\mathsf{Sat}(w, \delta, r)$, we can therefore pick a chain that repeatedly makes the same choice for $\delta$, and obtain a world along that chain, which is at the bar.

▶ **Example 28** (Beth Bars & References). Building up on Ex. 13, we present here an example where choices are references and bars as Beth bars, yielding an intuitionistic theory $\mathrm{TT}^{\mathsf{Beth}}_{\mathsf{Ref}}$ (see worldInstanceRef.lagda and modInstanceBethRef.lagda for details).

**Progress.** For Ref, $\mathsf{progress}(\delta, w_1, w_2)$ states that if a reference cell named $\delta$ holds $t$ in $w_1$, then it must also hold $t'$ in $w_2$, such that $t = t'$ if the cell is not mutable in $w_1$.

**Retrieving.** This property is trivial to prove for references because we need to exhibit a bar, which given $\delta \in \mathcal{N}$ and $n \in \mathbb{N}$, requires $\delta$'s $n^{th}$ choice to exist, which necessarily does because $\mathsf{choice?}(w, \delta, n)$ disregards its argument $n$ and returns $\delta$'s current content in $w$.

**Choice-following.** This property is proved as for free-choice sequences.

### 6.2.2 Open Bars

Open bars [5] are more straightforwardly defined and do not require the concept of chains.

▶ **Definition 29** (Open Bars). *Open bars are defined by the following bar predicate:* Open $:=$ $\lambda w.\lambda o.\forall^{\sqsubseteq}_{w}(w_1.\exists^{\sqsubseteq}_{w_1}(w_2.w_2 \in o))$*, which forms a bar space.*

The choice-following property does not hold for open bars due to the existential quantification in their definition, which allows different choices to be made. In fact, we can prove the negation of the choice-following property for open bars. Given $w_0 \in \mathcal{W}$, $\exists(\delta : \mathcal{N})(w : \mathcal{W})(P : \mathcal{P}_w)(r : \mathsf{Res}).\mathsf{Sat}(w, \delta, r) \wedge \Box_w P \wedge \forall^{\sqsubseteq}_{w}(w'.\mathsf{Sat}(w', \delta, r) \rightarrow \neg(P\ w'))$ holds by instantiating $\delta$ with $\nu\mathcal{C}(w_0)$, $w$ with $\mathsf{start}\nu\mathcal{C}(w_0, r)$, and $P$ with $\lambda w'.\neg\mathsf{mutable}(\delta, w')$, where $r$ restricts the choices to be either $\boldsymbol{\kappa_0}$ or $\boldsymbol{\kappa_1}$. Next we show that open bars induce a jumping modality, which is required to prove Lem. 20.

▶ **Example 30** (Open bars). The agnostic theory $\mathrm{TT}^{\mathsf{Open}}_{\mathcal{C}}$, built upon open bars and an arbitrary choice operator $\mathcal{C}$, is compatible with classical logic (see lem.lagda). In [5] this theory was presented specifically for Fcs. As choices are irrelevant to prove Lem. 20, we can

[4] To be precise, to prove that Beth bars satisfy the $\mathsf{non}\varnothing$ property, we further require a function $\mathit{ChofW}$ from $w \in \mathcal{W}$ to $\mathsf{chain}(w)$.

instantiate them with any suitable type, such as Ref or Fcs, and $\mathcal{W}$ can be any poset. It remains to show that Open satisfies the jumping property, which follows from the definition of open bars in terms of the existence of extensions of all extensions of the current world.

### 6.2.3 Kripke Bars

Let us present here another bar space, which allows capturing traditional Kripke semantics:

▶ **Definition 31** (Kripke Bars). *Kripke bars are defined by the following bar predicate:* Kripke $:= \lambda w.\lambda o.\forall_w^{\sqsubseteq}(w'.w' \in o)$*, which is a predicate that given a world $w$ requires opens to contain all extensions of $w$. This also forms a bar space as proved in barKripke.lagda.*

▶ **Example 32** (Kripke bars). According to Prop. 23, this space leads in turn to an equality modality, which captures traditional a Kripke semantics. However, as proved in kripkeCsNotRetrieving.lagda, this modality is not retrieving when choices are free-choice sequences, and therefore does not allow deriving the negation of classical axioms using Cor. 17. It is however retrieving when choices are references because reference cells are always filled with a value. We can then prove that the resulting equality modality along with references as choices satisfy all the properties required for Cor. 17 (see modInstanceKripkeRefBool.lagda). Therefore, the theory $\mathrm{TT}_{\mathsf{Ref}}^{\mathsf{Open}}$ is an intuitionistic theory, while $\mathrm{TT}_{\mathsf{Fcs}}^{\mathsf{Open}}$ is not.

## 7 Conclusions and Related Works

This paper provides a generic extensional type theory incorporating various time-progressing elements along with a possible-worlds forcing interpretation parameterized by modalities, which when instantiated with topological spaces of bars leads to a general sheaf model. We have opted for a general framework, both in terms of the choice operators it can embed and its modality-based semantics. This is so that our system is abstract enough to capture other general models from the literature, as well as for it to contain a wide class of theories, allowing us to reason collectively about their (in)compatibility with classical reasoning. Much remains to be explored to fully utilize our general framework to study the relation with classical reasoning. For one, the choice and modality properties presented in Sec. 5 provide sufficient conditions for determining the relation of the corresponding theories to classical reasoning. Further work is required to establish whether they are also necessary.

Other sheaf models for choice-like concepts have been proposed in the literature. We mention a few concrete examples that are most closely related to our general framework. In [19], the author provides a sheaf model of predicate logic extended with non-constructive objects such as choice sequences, where formulas are interpreted w.r.t. a forcing interpretation parameterized by a site. In [42], the authors provide sheaf models for the intuitionistic theories LS [38] and CS [27] featuring choice sequences, where formulas are essentially interpreted w.r.t. a forcing interpretation over the Baire space. In [12, 13], the authors prove the *uniform* continuity of a Martin-Löf-like intensional type theory using forcing, and extract an algorithm that computes a uniform modulus of continuity. In [23] the authors introduce a forcing translation for the Calculus of Inductive Constructions (CIC) [31] extended with effects, which crucially preserves definitional equality. In [14], the independence of MP with Martin-Löf's type theory is established through a forcing interpretation, with sequences of Booleans as forcing conditions, by following Brouwer's argument that it is not decidable whether a choice sequence of Booleans will remain true for ever or become eventually false.

Related to our work is also the line of work, starting from [33], on building syntactic models of CIC, by translating CIC extended with logical principles and effects into itself. Using this technique, in [7], the authors present syntactic models through which properties can be added to negative types, allowing them to prove independent results, e.g., the independence of function extensionality in intentional type theory. In [34], the authors present a translation, where the resulting type theory features exceptions, which is consistent if the target theory is when exceptions are required to be caught locally. The authors use this translation to exhibit syntactic models of CIC which validate the independence of premise axiom, but not MP. In [36], the authors solve the problem of the restriction on exceptions in [34] by introducing a layered type theory with exceptions, which separates the consistency and effectful programming concerns. In [32] the authors present a syntactic presheaf model of CIC, which solves issues with dependent elimination present in [23], and allows extending CIC with MP. In [35], the authors go back to these dependent elimination issues and present a new version of call-by-push-value which allows combining effects and dependent types.

Also connected to our work are the generic modal theories introduced in [21, 20]. In [21], the authors present a Martin-Löf type theory extended with an S4-style necessity modality, which satisfies normalization and decidability of type checking. To guarantee that the modality is an S4 necessity modality, this theory imposes restrictions on the terms inhabiting modalities, which are enforced through a “locking” mechanism. The generic modal type theory presented in [20] goes one step further from [21] by supporting multiple interacting modalities. Both theories share the goal of generically capturing hand-crafted modal theories, while we in particular focus on modalities “compatible” with choice operators.

## References


1. Agda wiki. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.
2. Stuart F. Allen. A non-type-theoretic definition of Martin-Löf’s types. In *LICS*, pages 215–221. IEEE Computer Society, 1987.
3. Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.
4. Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Computability beyond church-turing via choice sequences. In Anuj Dawar and Erich Grädel, editors, *LICS 2018*, pages 245–254. ACM, 2018. doi:10.1145/3209108.3209200.
5. Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Open bar - a brouwerian intuitionistic logic with a pinch of excluded middle. In Christel Baier and Jean Goubault-Larrecq, editors, *CSL*, volume 183 of *LIPIcs*, pages 11:1–11:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.11.
6. E. Bishop. *Foundations of constructive analysis*, volume 60. McGraw-Hill New York, 1967.
7. Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In Yves Bertot and Viktor Vafeiadis, editors, *CPP 2017*, pages 182–194. ACM, 2017. doi:10.1145/3018610.3018620.
8. Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987. URL: http://books.google.com/books?id=oN5nsPkXhhsC.
9. L. E. J Brouwer. Begründung der mengenlehre unabhängig vom logischen satz vom ausgeschlossen dritten. zweiter teil: Theorie der punkmengen. *Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, 12(7), 1919.
10. Paul J. Cohen. The independence of the continuum hypothesis. *the National Academy of Sciences of the United States of America*, 50(6):1143–1148, December 1963.
11. Paul J. Cohen. The independence of the continuum hypothesis ii. *the National Academy of Sciences of the United States of America*, 51(1):105–110, January 1964.

12 Thierry Coquand and Guilhem Jaber. A note on forcing and type theory. *Fundam. Inform.*, 100(1-4):43–52, 2010. doi:10.3233/FI-2010-262.

13 Thierry Coquand and Guilhem Jaber. A computational interpretation of forcing in type theory. In Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 203–213. Springer, 2012. doi:10.1007/978-94-007-4435-6_10.

14 Thierry Coquand and Bassel Mannaa. The independence of markov's principle in type theory. In Delia Kesner and Brigitte Pientka, editors, *FSCD 2016*, volume 52 of *LIPIcs*, pages 17:1–17:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FSCD.2016.17.

15 Karl Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

16 M. J. Cresswell and G. E. Hughes. *A New Introduction to Modal Logic*. Routledge, 1996.

17 Michael A. E. Dummett. *Elements of Intuitionism*. Clarendon Press, second edition, 2000.

18 VH Dyson and Georg Kreisel. *Analysis of Beth's semantic construction of intuitionistic logic*. Stanford University. Applied Mathematics and Statistics Laboratories, 1961.

19 Michael P. Fourman. Notions of choice sequence. In A.S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 91–105. Elsevier, 1982. doi:10.1016/S0049-237X(09)70125-9.

20 Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS*, pages 492–506. ACM, 2020. doi:10.1145/3373718.3394736.

21 Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.*, 3(ICFP):107:1–107:29, 2019. doi:10.1145/3341711.

22 Arend Heyting. *Intuitionism: an introduction*. North-Holland Pub. Co., 1956.

23 Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The definitional side of the forcing. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *LICS '16*, pages 367–376. ACM, 2016. doi:10.1145/2933575.2935320.

24 Stephen C. Kleene and Richard E. Vesley. *The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions*. North-Holland Publishing Company, 1965.

25 Georg Kreisel. A remark on free choice sequences and the topological completeness proofs. *J. Symb. Log.*, 23(4):369–388, 1958. doi:10.2307/2964012.

26 Georg Kreisel and Anne S. Troelstra. Formal systems for some branches of intuitionistic analysis. *Annals of Mathematical Logic*, 1(3):229–387, 1970. doi:10.1016/0003-4843(70)90001-X.

27 Georg Kreisel and Anne S. Troelstra. Formal systems for some branches of intuitionistic analysis. *Annals of mathematical logic*, 1(3):229–387, 1970.

28 Saul A. Kripke. Semantical analysis of modal logic i. normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9(5-6):67–96, 1963. doi:10.1002/malq.19630090502.

29 Saul A. Kripke. Semantical analysis of intuitionistic logic i. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, pages 92–130. Elsevier, 1965. doi:10.1016/S0049-237X(08)71685-9.

30 Joan R. Moschovakis. An intuitionistic theory of lawlike, choice and lawless sequences. In *Logic Colloquium'90: ASL Summer Meeting in Helsinki*, pages 191–209. Association for Symbolic Logic, 1993.

31 Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL: https://hal.inria.fr/hal-01094195.

32 Pierre-Marie Pédrot. Russian constructivism in a prefascist theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS*, pages 782–794. ACM, 2020. `doi:10.1145/3373718.3394740`.

33 Pierre-Marie Pédrot and Nicolas Tabareau. An effectful way to eliminate addiction to dependence. In *LICS 2017*, pages 1–12. IEEE Computer Society, 2017. `doi:10.1109/LICS.2017.8005113`.

34 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option - an exceptional type theory. In Amal Ahmed, editor, *ESOP 2018*, volume 10801 of *LNCS*, pages 245–271. Springer, 2018. `doi:10.1007/978-3-319-89884-1_9`.

35 Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020. `doi:10.1145/3371126`.

36 Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. A reasonably exceptional type theory. *Proc. ACM Program. Lang.*, 3(ICFP):108:1–108:29, 2019. `doi:10.1145/3341712`.

37 Andrew M Pitts. Nominal sets: Names and symmetry in computer science, volume 57 of cambridge tracts in theoretical computer science, 2013.

38 Anne S. Troelstra. *Choice sequences: a chapter of intuitionistic mathematics.* Clarendon Press Oxford, 1977.

39 Anne S. Troelstra. Choice sequences and informal rigour. *Synthese*, 62(2):217–227, 1985.

40 Mark van Atten and Dirk van Dalen. Arguments for the continuity principle. *Bulletin of Symbolic Logic*, 8(3):329–347, 2002. URL: `http://www.math.ucla.edu/~asl/bsl/0803/0803-001.ps`, `doi:10.2178/bsl/1182353892`.

41 Dirk van Dalen. An interpretation of intuitionistic analysis. *Annals of mathematical logic*, 13(1):1–43, 1978.

42 Gerrit Van Der Hoeven and Ieke Moerdijk. Sheaf models for choice sequences. *Annals of Pure and Applied Logic*, 27(1):63–107, 1984. `doi:10.1016/0168-0072(84)90035-6`.

43 Wim Veldman. Understanding and using Brouwer's continuity principle. In *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*, volume 306 of *Synthese Library*, pages 285–302. Springer Netherlands, 2001. `doi:10.1007/978-94-015-9757-9_24`.

44 Beth E. W. Semantic construction of intuitionistic logic. *Journal of Symbolic Logic*, 22(4):363–365, 1957.


# A $\mathsf{TT}_{\mathcal{C}}^{\square}$'s Inference Rules

In $\mathrm{TT}_{\mathcal{C}}^{\square}$, sequents are of the form $h_1, \ldots, h_n \vdash t : T$. Such a sequent denotes that, assuming $h_1, \ldots, h_n$, the term $t$ is a member of the type $T$, and that therefore $T$ is a type. The term $t$ in this context is called the *extract* of $T$. Extracts are sometimes omitted when irrelevant to the discussion. An hypothesis $h$ is of the form $x{:}A$, where the variable $x$ stands for the name of the hypothesis and $A$ its type. A rule is a pair of a conclusion sequent $S$ and a list of premise sequents, $S_1, \cdots, S_n$ (written as usual using a fraction notation, with the premises on top). Let us now provide a sample of $\mathrm{TT}_{\mathcal{C}}^{\square}$'s key inference rules for some of its types not discussed above. In what follows, we write $a{\in}A$ for $a{=}a{\in}A$.

**Products.** The following rules are the standard $\mathbf{\Pi}$-elimination rule, $\mathbf{\Pi}$-introduction rule, type equality for $\mathbf{\Pi}$ types, and $\lambda$-introduction rule, respectively.

$$\frac{H, f{:}\mathbf{\Pi}x{:}A.B, J \vdash a{\in}A \quad H, f{:}\mathbf{\Pi}x{:}A.B, J, z{:}f(a){\in}B[x\backslash a] \vdash e : C}{H, f{:}\mathbf{\Pi}x{:}A.B, J \vdash e[z\backslash\star] : C} \qquad \frac{H, z{:}A \vdash b : B[x\backslash z] \quad H \vdash A{\in}\mathbb{U}_i}{H \vdash \lambda z.b : \mathbf{\Pi}x{:}A.B}$$

$$\frac{H \vdash A_1{=}A_2{\in}\mathbb{U}_i \quad H, y{:}A_1 \vdash B_1[x_1\backslash y]{=}B_2[x_2\backslash y]{\in}\mathbb{U}_i}{H \vdash \mathbf{\Pi}x_1{:}A_1.B_1{=}\mathbf{\Pi}x_2{:}A_2.B_2{\in}\mathbb{U}_i} \qquad \frac{H, z{:}A \vdash t_1[x_1\backslash z]{=}t_2[x_2\backslash z]{\in}B[x\backslash z] \quad H \vdash A{\in}\mathbb{U}_i}{H \vdash \lambda x_1.t_1{=}\lambda x_2.t_2{\in}\mathbf{\Pi}x{:}A.B}$$

Note that the last rule requires to prove that $A$ is a type because the conclusion requires to prove that $\mathbf{\Pi}x{:}A.B$ is a type, and the first hypothesis only states that $B$ is a type family over $A$, but does not ensures that $A$ is a type. Furthermore, the following rules are the standard function extensionality and $\beta$-reduction rules, respectively:

$$\frac{H, z{:}A \vdash f_1(z){=}f_2(z){\in}B[x\backslash z] \quad H \vdash A{\in}\mathbb{U}_i}{H \vdash f_1{=}f_2{\in}\mathbf{\Pi}x{:}A.B} \qquad \frac{H \vdash t[x\backslash s]{=}u{\in}T}{H \vdash (\lambda x.t)\ s{=}u{\in}T}$$

**Sums.** The following rules are the standard $\mathbf{\Sigma}$-elimination rule, $\mathbf{\Sigma}$-introduction rule, type equality for the $\mathbf{\Sigma}$ type, pair-introduction, and spread-reduction rules, respectively:

$$\frac{H, p{:}\mathbf{\Sigma}x{:}A.B, a{:}A, b{:}B[x\backslash a], J[p\backslash\langle a,b\rangle] \vdash e : C[p\backslash\langle a,b\rangle]}{H, p{:}\mathbf{\Sigma}x{:}A.B, J \vdash \texttt{let}\ a,b = p\ \texttt{in}\ e : C} \qquad \frac{H \vdash a{\in}A \quad H \vdash b{\in}B[x\backslash a] \quad H, z{:}A \vdash B[x\backslash z]{\in}\mathbb{U}_i}{H \vdash \langle a,b\rangle : \mathbf{\Sigma}x{:}A.B}$$

$$\frac{H \vdash A_1{=}A_2{\in}\mathbb{U}_i \quad H, y{:}A_1 \vdash B_1[x_1\backslash y]{=}B_2[x_2\backslash y]{\in}\mathbb{U}_i}{H \vdash \mathbf{\Sigma}x_1{:}A_1.B_1{=}\mathbf{\Sigma}x_2{:}A_2.B_2{\in}\mathbb{U}_i} \qquad \frac{H, z{:}A \vdash B[x\backslash z]{\in}\mathbb{U}_i \quad H \vdash a_1{=}a_2{\in}A \quad H \vdash b_1{=}b_2{\in}B[x\backslash a_1]}{H \vdash \langle a_1,b_1\rangle{=}\langle a_2,b_2\rangle{\in}\mathbf{\Sigma}x{:}A.B}$$

$$\frac{H \vdash u[x\backslash s; y\backslash t]{=}t_2{\in}T}{H \vdash \texttt{let}\ x,y = \langle s,t\rangle\ \texttt{in}\ u{=}t_2{\in}T}$$

**Equality.** The following rules are the standard equality-introduction rule, equality-elimination rule, hypothesis rule, symmetry and transitivity rules, respectively.

$$\frac{H \vdash A{=}B{\in}\mathbb{U}_i \quad H \vdash a_1{=}b_1{\in}A \quad H \vdash a_2{=}b_2{\in}B}{H \vdash (a_1{=}a_2{\in}A){=}(b_1{=}b_2{\in}B){\in}\mathbb{U}_i} \qquad \frac{H, z{:}a{=}b{\in}A, J[z\backslash\star] \vdash e : C[z\backslash\star]}{H, z{:}a{=}b{\in}A, J \vdash e : C}$$

$$\frac{}{H, x{:}A, J \vdash x{\in}A} \qquad \frac{H \vdash b{=}a{\in}T}{H \vdash a{=}b{\in}T} \qquad \frac{H \vdash a{=}c{\in}T \quad H \vdash c{=}b{\in}T}{H \vdash a{=}b{\in}T}$$

The following rules allow fixing the extract of a sequent, and rewriting with an equality in an hypothesis, respectively:

$$\frac{H \vdash t : T}{H \vdash t{\in}T} \qquad \frac{H, x{:}B, J \vdash t : C \quad H \vdash A{=}B{\in}\mathbb{U}_i}{H, x{:}A, J \vdash t : C}$$

**Universes.** Let $i$ be a lower universe than $j$. The following rules are the standard universe-introduction rule and the universe cumulativity rule, respectively.

$$\frac{}{H \vdash \mathbb{U}_i{=}\mathbb{U}_i{\in}\mathbb{U}_j} \qquad \frac{H \vdash T{\in}\mathbb{U}_j}{H \vdash T{\in}\mathbb{U}_i}$$

**Sets.** The following rule is the standard set-elimination rule:

$$\frac{H, z{:}\{x : A \mid B\}, a{:}A, \boxed{b{:}B[x\backslash a]}, J[z\backslash a] \vdash e : C[z\backslash a]}{H, z{:}\{x : A \mid B\}, J \vdash e[a\backslash z] : C}$$

Note that we have used a new construct in the above rule: the *hidden* hypothesis $\boxed{b{:}B[x\backslash a]}$. The main feature of hidden hypotheses is that their names cannot occur in extracts (which is why we "box" those hypotheses). Intuitively, this is because the proof that $B$ is true is discarded in the proof that the set type $\{x : A \mid B\}$ is true and therefore cannot occur in computations. Hidden hypotheses can be unhidden using the following rule:

$$\frac{H, x{:}T, J \vdash \star : a{=}b{\in}A}{H, \boxed{x{:}T}, J \vdash \star : a{=}b{\in}A}$$

which is valid since the extract is $\star$ and therefore does not make use of $x$.

The following rules are the standard set-introduction rule, type equality for the set type, and introduction rule for members of set types, respectively.

$$\frac{H \vdash a \in A \quad H \vdash B[x\backslash a] \quad H, z{:}A \vdash B[x\backslash z] \in \mathbb{U}_i}{H \vdash a : \{x : A \mid B\}} \qquad \frac{H \vdash A_1 = A_2 \in \mathbb{U}_i \quad H, y{:}A_1 \vdash B_1[x_1\backslash y] = B_2[x_2\backslash y] \in \mathbb{U}_i}{H \vdash \{x_1 : A_1 \mid B_1\} = \{x_2 : A_2 \mid B_2\} \in \mathbb{U}_i}$$

$$\frac{H, z{:}A \vdash B[x\backslash z] \in \mathbb{U}_i \quad H \vdash a = b \in A \quad H \vdash B[x\backslash a]}{H \vdash a = b \in \{x : A \mid B\}}$$

**Disjoint Unions.** The following rules are the disjoint union-elimination, disjoint union-introduction (left and right), type equality for disjoint unions, injection-introduction (left and right), and decide-reduction (left and right) rules, respectively:

$$\frac{H, d{:}A{+}B, x{:}A, J[d\backslash \mathtt{inl}(x)] \vdash t : C[d\backslash \mathtt{inl}(x)] \qquad H, d{:}A{+}B, y{:}B, J[d\backslash \mathtt{inr}(y)] \vdash u : C[d\backslash \mathtt{inr}(y)]}{H, d{:}A{+}B, J \vdash \mathtt{case}\ d\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t \mid \mathtt{inr}(y) \Rightarrow u : C}$$

$$\frac{H \vdash a : A \quad H \vdash B \in \mathbb{U}_i}{H \vdash \mathtt{inl}(a) : A{+}B} \qquad \frac{H \vdash b : B \quad H \vdash A \in \mathbb{U}_i}{H \vdash \mathtt{inr}(b) : A{+}B} \qquad \frac{H \vdash A_1 = A_2 \in \mathbb{U}_i \quad H \vdash B_1 = B_2 \in \mathbb{U}_i}{H \vdash A_1{+}B_1 = A_2{+}B_2 \in \mathbb{U}_i}$$

$$\frac{H \vdash a_1 = a_2 \in A \quad H \vdash B \in \mathbb{U}_i}{H \vdash \mathtt{inl}(a_1) = \mathtt{inl}(a_2) \in A{+}B} \qquad \frac{H \vdash b_1 = b_2 \in B \quad H \vdash A \in \mathbb{U}_i}{H \vdash \mathtt{inr}(b_1) = \mathtt{inr}(b_2) \in A{+}B}$$

$$\frac{H \vdash t[x\backslash s] = t_2 \in T}{H \vdash (\mathtt{case}\ \mathtt{inl}(s)\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t \mid \mathtt{inr}(y) \Rightarrow u) = t_2 \in T} \qquad \frac{H \vdash u[y\backslash s] = t_2 \in T}{H \vdash (\mathtt{case}\ \mathtt{inr}(s)\ \mathtt{of}\ \mathtt{inl}(x) \Rightarrow t \mid \mathtt{inr}(y) \Rightarrow u) = t_2 \in T}$$

**Time-Quotients.** The following rules are the introduction and type equality rules for the time-quotienting type. Note that in practice more terms than the ones in $A$ can be shown to be in $\wr A$. For example, given a choice name $\delta$ with a restriction that constrains its choices to be elements of $A$, we can prove that $\delta(\underline{n})$, for $n \in \mathbb{N}$ is in $\wr A$, even though $\delta(\underline{n})$ might change over time. Devising such rules, as well as elimination rules, is left for future work.

$$\frac{H \vdash a : A}{H \vdash a : \wr A} \qquad \frac{H \vdash A = B \in \mathbb{U}_i}{H \vdash \wr A = \wr B \in \mathbb{U}_i} \qquad \frac{H \vdash a = b \in A}{H \vdash a = b \in \wr A}$$

# B Equality Modalities

As mentioned in Sec. 4, our forcing interpretation relies on a pair of a modality $\square$ and a dependent modality $\boxdot$. The version of this interpretation presented there is a consequence of the formal definition, which involves both modalities. Let us now describe this definition in this section (see forcing.lagda for further details). We define in Fig. 3 an $w \vDash_l T_1 \equiv T_2$ set, which compared to the one presented in Sec. 4, contains a universe level annotation $l$, which is simply here a $\mathbb{N}$. In addition, that figure defines a recursive function $w \vDash_l a \equiv b \in e$, which recurses over $e \in w \vDash_l T_1 \equiv T_2$, and again contains a universe level annotation compared to the one presented in Sec. 4. This inductive-recursive definition is defined recursively over universe levels. The function $w \vDash a \equiv b \in T$ presented in Sec. 4 can then be defined as $\exists(l : \mathbb{N})(e : w \vDash_l T \equiv T).w \vDash a \equiv b \in e$.

Let us now formally introduce the dependent modality $\boxdot^i_w$, along with its properties. First, we introduce a dependent version of the set $\mathcal{P}_w$ as follows: the collection of predicates in $\forall w' \sqsupseteq w.P\ w' \to \mathbb{P}$ for $P \in \mathcal{P}_w$, is denoted $\mathcal{P}^P_w$. The dependent modality $\boxdot^i_w \in \mathcal{P}^P_w \to \mathbb{P}$, where $P \in \mathcal{P}_w \to \mathbb{P}$ and $i \in \square_w P$, is called a *dependent equality modality*

Note that as for members of $\mathcal{P}_w$, due to $\sqsubseteq$'s transitivity, if $Q \in \mathcal{P}^P_w$, where $P \in \mathcal{P}_w$, then for every $w' \sqsupseteq w$, it naturally extends to a predicate in $\mathcal{P}^P_{w'}$. Also, note that property $\square_1$ in Def. 8 can be viewed as defining a lifting operator $\uparrow_{w'} i$, which returns a $\square_{w'} P$, given a $w' \sqsupseteq w$ and $i \in \square_w P$ as specified there. This lifting operator will be used to state $\boxdot^i_w$'s properties.

We can now state $\boxdot^i_w$'s properties, which are counterparts of properties $\square_1, \square_2, \square_3$:

Inductive relation:

$$
\begin{aligned}
w \vDash_l T_1 \equiv T_2 ::= \ & \mathtt{NAT}\equiv(T_1 \Downarrow_w \mathsf{Nat} \wedge T_2 \Downarrow_w \mathsf{Nat})\\
& \mid \mathtt{PI}\equiv\left(\begin{array}{l}\exists(x:\mathsf{Var})(A_1,A_2,B_1,B_2:\mathsf{Term})(e:\forall^{\sqsubseteq}_w(w'.w'\vDash_l A_1\equiv A_2)).\\ \quad T_1 \Downarrow_w \mathbf{\Pi} x{:}A_1.B_1 \wedge T_2 \Downarrow_w \mathbf{\Pi} x{:}A_2.B_2\\ \quad \wedge \forall^{\sqsubseteq}_w(w'.\forall(a,b:\mathsf{Term}).w'\vDash_l a\equiv b\in(e\ w') \to w'\vDash_l B_1[x\backslash a]\equiv B_2[x\backslash b])\end{array}\right)\\
& \mid \mathtt{SUM}\equiv\left(\begin{array}{l}\exists(x:\mathsf{Var})(A_1,A_2,B_1,B_2:\mathsf{Term})(e:\forall^{\sqsubseteq}_w(w'.w'\vDash_l A_1\equiv A_2)).\\ \quad T_1 \Downarrow_w \mathbf{\Sigma} x{:}A_1.B_1 \wedge T_2 \Downarrow_w \mathbf{\Sigma} x{:}A_2.B_2\\ \quad \wedge \forall^{\sqsubseteq}_w(w'.\forall(a,b:\mathsf{Term}).w'\vDash_l a\equiv b\in(e\ w') \to w'\vDash_l B_1[x\backslash a]\equiv B_2[x\backslash b])\end{array}\right)\\
& \mid \mathtt{SET}\equiv\left(\begin{array}{l}\exists(x:\mathsf{Var})(A_1,A_2,B_1,B_2:\mathsf{Term})(e:\forall^{\sqsubseteq}_w(w'.w'\vDash_l A_1\equiv A_2)).\\ \quad T_1 \Downarrow_w \{x:A_1 \mid B_1\} \wedge T_2 \Downarrow_w \{x:A_2 \mid B_2\}\\ \quad \wedge \forall^{\sqsubseteq}_w(w'.\forall(a,b:\mathsf{Term}).w'\vDash_l a\equiv b\in(e\ w') \to w'\vDash_l B_1[x\backslash a]\equiv B_2[x\backslash b])\end{array}\right)\\
& \mid \mathtt{UNION}\equiv\left(\begin{array}{l}\exists(A_1,A_2,B_1,B_2:\mathsf{Term}).T_1 \Downarrow_w A_1{+}B_1 \wedge T_2 \Downarrow_w A_2{+}B_2\\ \quad \wedge \forall^{\sqsubseteq}_w(w'.w'\vDash_l A_1\equiv A_2) \wedge \forall^{\sqsubseteq}_w(w'.w'\vDash_l B_1\equiv B_2)\end{array}\right)\\
& \mid \mathtt{EQ}\equiv\left(\begin{array}{l}\exists(a_1,a_2,b_1,b_2,A,B:\mathsf{Term})(e:\forall^{\sqsubseteq}_w(w'.w'\vDash_l A\equiv B)).\\ \quad T_1 \Downarrow_w a_1{=}a_2{\in}A \wedge T_2 \Downarrow_w b_1{=}b_2{\in}B\\ \quad \wedge \forall^{\sqsubseteq}_w(w'.w'\vDash_l a_1\equiv b_1\in(e\ w')) \wedge \forall^{\sqsubseteq}_w(w'.w'\vDash_l a_2\equiv b_2\in(e\ w'))\end{array}\right)\\
& \mid \mathtt{QTIME}\equiv(\exists(A,B:\mathsf{Term}).T_1 \Downarrow_w \$A \wedge T_2 \Downarrow_w \$B \wedge \forall^{\sqsubseteq}_w(w'.w'\vDash_l A\equiv B))\\
& \mid \mathtt{MOD}\equiv(\square_w(w'.w'\vDash_l T_1\equiv T_2))\\
& \mid \mathtt{UNIV}\equiv(\exists(j<l).T_1 \Downarrow_w \mathbb{U}_j \wedge T_2 \Downarrow_w \mathbb{U}_j)
\end{aligned}
$$

Recursive function:

$$
\begin{aligned}
& w \vDash_l t\equiv t'\in\mathtt{NAT}\equiv(c_1,c_2) := \square_w(w'.\exists(n:\mathbb{N}).t \Downarrow_{w'} \underline{n} \wedge t' \Downarrow_{w'} \underline{n})\\
& w \vDash_l t\equiv t'\in\mathtt{PI}\equiv(x,A_1,A_2,B_1,B_2,e,c_1,c_2,f)\\
& \quad := \square_w(w'.\forall(a_1,a_2:\mathsf{Term})(i:w'\vDash_l a_1\equiv a_2\in(e\ w')).w'\vDash_l (t\ a_1)\equiv(t'\ a_2)\in(f\ w'\ a_1\ a_2\ i))\\
& w \vDash_l t\equiv t'\in\mathtt{SUM}\equiv(x,A_1,A_2,B_1,B_2,e,c_1,c_2,f)\\
& \quad := \square_w\left(w'.\begin{array}{l}\exists(a_1,a_2,b_1,b_2:\mathsf{Term})(i:w'\vDash_l a_1\equiv a_2\in(e\ w')).\\ \quad w'\vDash_l b_1\equiv b_2\in(f\ w'\ a_1\ a_2\ i) \wedge t \Downarrow_{w'} \langle a_1,b_1\rangle \wedge t' \Downarrow_{w'} \langle a_2,b_2\rangle\end{array}\right)\\
& w \vDash_l t\equiv t'\in\mathtt{SET}\equiv(x,A_1,A_2,B_1,B_2,e,c_1,c_2,f)\\
& \quad := \square_w(w'.\exists(b_1,b_2:\mathsf{Term})(i:w'\vDash_l t\equiv t'\in(e\ w')).w'\vDash_l b_1\equiv b_2\in(f\ w'\ t\ t'\ i))\\
& w \vDash_l t\equiv t'\in\mathtt{UNION}\equiv(A_1,A_2,B_1,B_2,c_1,c_2,e,f)\\
& \quad := \square_w\left(w'.\exists(u,v:\mathsf{Term}).\begin{array}{l}(t \Downarrow_{w'} \mathtt{inl}(u) \wedge t' \Downarrow_{w'} \mathtt{inl}(v) \wedge w'\vDash_l u\equiv v\in(e\ w'))\\ \vee\ (t \Downarrow_{w'} \mathtt{inr}(u) \wedge t' \Downarrow_{w'} \mathtt{inr}(v) \wedge w'\vDash_l u\equiv v\in(f\ w'))\end{array}\right)\\
& w \vDash_l t\equiv t'\in\mathtt{EQ}\equiv(a_1,a_2,b_1,b_2,A,B,e,c_1,c_2,i_1,i_2) := \square_w(w'.w'\vDash_l a_1\equiv a_2\in(e\ w'))\\
& w \vDash_l t\equiv t'\in\mathtt{QTIME}\equiv(A,B,c_1,c_2,e) := \square_w(w'.(\lambda a,b.\exists(c,d:\mathsf{Value}).a \sim_w c \wedge b \sim_w d \wedge w \vDash_l c\equiv d\in(e\ w'))^+\ t\ t')\\
& w \vDash_l t\equiv t'\in\mathtt{MOD}\equiv(i) := \boxdot^i_w(w'.\lambda(j:w'\vDash_l T_1\equiv T_2).w'\vDash_l t\equiv t'\in j), \text{where } i \text{ is a proof of } \square_w(w'.w'\vDash_l T_1\equiv T_2)\\
& w \vDash_l t\equiv t'\in\mathtt{UNIV}\equiv(j,c_1,c_2) := w \vDash_j t\equiv t', \text{where } j<l
\end{aligned}
$$

**Figure 3** Inductive-Recursive Forcing Interpretation.

- $\boxdot_1$: monotonicity of $\boxdot$: $\forall(w:\mathcal{W})(P:\mathcal{P}_w)(Q:\mathcal{P}^P_w)(i:\square_w P).\forall w' \sqsupseteq w.\ \boxdot^i_w Q \to \boxdot^{\uparrow_{w'} i}_{w'} Q$. This property defines a lifting operator $\uparrow_{w'} j$, which returns a $\boxdot^{\uparrow_{w'} i}_{w'} Q$, given a $w' \sqsupseteq w$ and $j \in \boxdot^i_w Q$ as specified above.
- $\boxdot_2$: A version of the distribution axiom:

$$
\begin{aligned}
&\forall(w:\mathcal{W})(P_1,P_2,P_3:\mathcal{P}_w)(Q_1:\mathcal{P}^{P_1}_w)(Q_2:\mathcal{P}^{P_2}_w)(Q_3:\mathcal{P}^{P_3}_w)(i_1:\square_w P_1)(i_2:\square_w P_2)(i_3:\square_w P_3).\\
&\quad (\forall^{\sqsubseteq}_w(w')\forall(p_1:P_1\ w')(p_2:P_2\ w')(p_3:P_3\ w').Q_1\ w'\ p_1 \to Q_2\ w'\ p_2 \to Q_3\ w'\ p_3)\\
&\quad \to \boxdot^{i_1}_w Q_1 \to \boxdot^{i_2}_w Q_2 \to \boxdot^{i_3}_w Q_3
\end{aligned}
$$

- $\boxdot_3$: $\boxdot$ follows from $\square\boxdot$, i.e., a dependent version of C4:

$$\forall(w:\mathcal{W})(P:\mathcal{P}_w)(Q:\mathcal{P}^P_w)(i:\square_w P).\ \square_w\,(w'.\ \boxdot^{\uparrow_{w'} i}_{w'}\ Q) \to \boxdot^i_w Q$$

In addition, the two modalities $\square$ and $\boxdot$ are required to satisfy the following properties that allow deriving one from other in some contexts, namely that $\boxdot$ follows from $\square$ and $\square$ follows from $\boxdot$, respectively:

- $\forall(w:\mathcal{W})(P:\mathcal{P}_w)(Q:\mathcal{P}^P_w).\ \square_w\,(w'.\forall(p:P\ w').Q\ w'\ p) \to \forall(i:\square_w P).\ \boxdot^i_w\,Q$
- $\forall(w:\mathcal{W})(P,R:\mathcal{P}_w)(Q:\mathcal{P}^P_w)(i:\square_w P).\forall^{\sqsubseteq}_w(w')\forall(p:P\ w').Q\ w'\ p \to R\ w' \to \boxdot^i_w Q \to \square_w R$

## C Properties of the Bar Space

The properties of bar spaces presented in Def. 22 allow deriving corresponding bars as follows:

- Intersection of bars. Given a bar predicate $B \in \mathsf{BarProp}$ such that $\mathsf{isect}(B)$, and two bars $b_1, b_2 \in \mathcal{B}^w_B$ for some world $w$, then $b_1 \cap b_2 \in \mathcal{B}^w_B$: $w \triangleleft (b_1 \cap b_2) \in B$ follows from $\mathsf{isect}(B)$; the two other properties of bars follow from the definition of $b_1 \cap b_2$.
- Union of bars. Given a bar predicate $B \in \mathsf{BarProp}$ such that $\mathsf{union}(B)$, and a family of bars $i \in \forall w' \sqsupseteq w.w' \in b \to \mathcal{B}^{w'}_B$ for some world $w$, then $\cup(i) \in \mathcal{B}^w_B$: $w \triangleleft (\cup(i)) \in B$ follows from $\mathsf{union}(B)$; the two other properties of bars follow from the definition of $\cup(i)$.
- Top bar. Given a bar predicate $B \in \mathsf{BarProp}$, such that $\mathsf{top}(B)$, then $\top(w) \in \mathcal{B}^w_B$: $w \triangleleft (\top(w)) \in B$ follows from $\mathsf{top}(B)$; the two other properties of bars follow from the definition of $\top(w)$.
- Sub-bar. Given a bar predicate $B \in \mathsf{BarProp}$ such that $\mathsf{sub}(B)$, and a bar $b \in \mathcal{B}^w_B$ for some world $w$, then $b\!\downharpoonright_{w'} \in \mathcal{B}^{w'}_B$ for any $w' \sqsupseteq w$: $w \triangleleft (b\!\downharpoonright_{w'}) \in B$ follows from $\mathsf{sub}(B)$; the two other properties of bars follow from the definition of $b\!\downharpoonright_{w'}$.

As mentioned in Prop. 23, $\exists\mathcal{B}^w_B$, where $B \in \mathsf{BarSpace}$ and $w \in \mathcal{W}$, is an equality modality. We can derive the properties (see Def. 8) of this modality as follows:

- To prove $\square_1$, we need to derive $\exists\mathcal{B}^{w'}_B(P)$ from $\exists\mathcal{B}^w_B(P)$, where $w' \sqsupseteq w$. As $\exists\mathcal{B}^w_B(P)$ gives us a bar $b \in \mathcal{B}^w_B$, we can instantiate our conclusion with $b\!\downharpoonright_{w'}$.
- To prove $\square_2$, we need to derive $\exists\mathcal{B}^w_B(Q)$ from $\exists\mathcal{B}^w_B(\lambda w'.P\ w' \to Q\ w')$ and $\exists\mathcal{B}^w_B(P)$. Our first assumption gives us a bar $b_1 \in \mathcal{B}^w_B$ and our second assumption gives us a bar $b_2 \in \mathcal{B}^w_B$. We can then instantiate our conclusion with $b_1 \cap b_2$.
- To prove $\square_3$, we need to derive $\exists\mathcal{B}^w_B(P)$ from $\exists\mathcal{B}^w_B(\lambda w'.\exists\mathcal{B}^{w'}_B(P))$. This assumption gives us a bar $b \in \mathcal{B}^w_B$ along with a function $i \in (\lambda w'.\exists\mathcal{B}^{w'}_B(P)) \in b$. We can then instantiate our conclusion with $\cup(i)$.
- To prove $\square_4$, we need to derive $\exists\mathcal{B}^w_B(P)$ from $\forall^{\sqsubseteq}_w(P)$. We can then instantiate our conclusion using $\top(w)$, and have to prove $P \in \top(w)$, which trivially follows from $\forall^{\sqsubseteq}_w(P)$.
- To prove $\square_5$, we need to derive $P$ from $\exists\mathcal{B}^w_B(\lambda w'.P)$. This assumption gives us a bar $b$ such that $(\lambda w'.P) \in b$. From $\mathsf{non}\varnothing(B)$, we obtain a $w' \sqsupseteq w$ such that $w' \in b$. We can then instantiate $(\lambda w'.P) \in b$ with $w'$, and we obtain $P$ since it does not depend on a world.

# Division by Two, in Homotopy Type Theory

**Samuel Mimram**
École polytechnique, Palaiseau, France

**Émile Oleon**
École polytechnique, Palaiseau, France

## Abstract

Natural numbers are isomorphism classes of finite sets and one can look for operations on sets which, after quotienting, allow recovering traditional arithmetic operations. Moreover, from a constructivist perspective, it is interesting to study whether those operations can be performed without resorting to the axiom of choice (the use of classical logic is usually necessary). Following the work of Bernstein, Sierpiński, Doyle and Conway, we study here "division by two" (or, rather, regularity of multiplication by two). We provide here a full formalization of this operation on sets, using the cubical variant of Agda, which is an implementation of the homotopy type theory setting, thus revealing some interesting points in the proof. As a novel contribution, we also show that this construction extends to general types, as opposed to sets.

**2012 ACM Subject Classification** Theory of computation → Constructive mathematics

**Keywords and phrases** division, axiom of choice, set theory, homotopy type theory, Agda

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.11

**Supplementary Material** `https://github.com/smimram/div2`

## 1 Introduction

**Dividing sets without choice.** Natural numbers can be defined as equivalence classes of finite sets under isomorphism: they allow one to count the number of elements of a finite set. Taking this point of view, it is natural to ask whether the usual operations on natural numbers are quotients of reasonable corresponding operations on sets, which would moreover generalize to infinite sets: these operations on sets have the advantage of being more explicit, in the sense that we produce a bijection instead of a mere equality. For instance, it is clear that addition and multiplication of natural numbers respectively correspond to disjoint union and cartesian product of sets. Namely, writing $|A|$ for the cardinal of a finite set $A$, i.e. its equivalence class under isomorphism, we have $|A \sqcup B| = |A| + |B|$ and $|A \times B| = |A| \times |B|$. This process of finding operations which correspond to already known ones after quotienting is also known as *categorification* in the context of (higher) category theory.

The next operation one might be tempted to implement is subtraction by 1 or *predecessor* function (subtraction by a finite number can of course be obtained by iterating it). Since predecessor of zero is not defined, we rather want to show that successor is *regular*, i.e. that $m+1 = n+1$ implies $m = n$. In terms of sets, this means that from a bijection $A \sqcup 1 \simeq B \sqcup 1$, we should be able to construct a bijection $A \simeq B$, where 1 denotes any set with one element: this is easily performed (and detailed in Section 2). Similarly, one can try to construct "division by 2": given natural numbers $m$ and $n$, we want to show that $m \times 2 = n \times 2$ implies $m = n$. In terms of sets, this means that from a bijection $A \times 2 \simeq B \times 2$, we should be able to construct a bijection $A \simeq B$ (where, of course, 2 denotes any set with two elements). Well, again, this is easily performed: if the sets are finite, we are essentially in the setting of natural numbers, and if the sets are infinite we have $A \simeq A \sqcup A \simeq B \sqcup B \simeq B$. Case settled.

© Samuel Mimram and Émile Oleon;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 11; pp. 11:1–11:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, a *constructivist* will immediately notice that we have used two debatable principles in the previous reasoning: the excluded middle (any set is finite or not) and the axiom of choice (in order to construct a bijection $A \simeq A \sqcup A$ when $A$ is infinite). It is thus natural to ask whether such an operation can still be performed in a more constructive setting, i.e. without resorting to one or both principles. Such questions can be traced back to the early 20th century: Bernstein gave a proof in his PhD thesis in 1901 [1] (see also [7, chapter 14]) that division by 2 could be performed in classical Zermelo-Fraenkel set theory without the axiom of choice (ZF). His proof was later much simplified by Sierpiński in 1922 [16]. The generalization to division by a finite cardinal was apparently solved in 1926 by Lindenbaum and Tarski [11], but their solution was not published and got forgotten, and Tarski found in 1949 a new solution to the problem which, this time, was published [18]. While working on this problem, Conway and Doyle managed to reconstruct what they believe is Lindenbaum and Tarski's original proof [3]. This article, which we discovered after the recent death of the first author, was our first introduction to the subject, and we mostly follow the proof given there: it is written in a delightful semi-formal way and we urge the reader to have a look at it. Since then, the construction of the division was refined and simplified [4, 15] and variants were explored [12].

As mentioned earlier, most efforts were concentrated on constructing division without the axiom of choice, but one can also wonder whether the excluded middle is necessary. The answer is unfortunately positive: this was shown in [17] by exhibiting a non-boolean topos in which multiplication by 2 is not regular.

**Formalizing division by two.** In this paper, we present a full formalization, in the Agda proof assistant, of division by 2, closely following Conway and Doyle's proof [3], whose code is publicly available [13]. Before going any further, let us first answer the obvious question: why would we want to do such a thing?

A first reason is to make sure that the results do actually hold. While there is no particular reason to have doubts about the validity of the constructions and associated proofs, the two primary sources [16, 3] are respectively written in a very concise way and in an informal way and it is reassuring to have a fully detailed proof, especially since it is easy to unknowingly use a non-constructive principle such as the axiom of choice. Moreover the point of being constructive is precisely to be able to construct thing (or, more precisely, programs), which we put in application here. Finally, detailing the proof, enables one to formulate interesting conjectures and opens research tracks.

The formalization is performed in the recent *cubical* variant of Agda [20] which is based on the interpretation of homotopy type theory developed by Coquand and collaborators [2]. The primary reason is that it offers the possibility of defining *higher inductive types* or *HITs* (those are like regular inductive types where equalities can freely be added) such as propositional truncation, quotients or integers, which we will see allow us to elegantly express the concepts required in order to formalize our proof. This setting validates the *univalence axiom*, meaning that we actually work in *homotopy* type theory or $HoTT$ [19]. Most of the types we use are however actually sets (contrarily to what we first hoped, see below), and this development shows that HoTT can be very relevant for the formalization of set-theoretic results: in addition to bringing in HITs, as mentioned above, it also allows transporting elements of dependent types along equalities, in a computational way, and we make much use of this here. We believe that this development also serves as a good illustration that cubical Agda and the associated library are mature enough to formalize some non-trivial properties in traditional (set-theoretic) mathematics.

The proof we provide roughly takes 3000 lines of Agda, whereas it takes roughly 6 pages both in [16] and in [3, section 5] (if we exclude full-page hand-drawn figures), which should give the reader a good idea of how many statements are left implicit in usual proofs. The reason why we stopped at 2 and did not formalize division by 3 (or the more general case of dividing by a finite set, which is close) is that, while the ideas involved in the construction are difficult to come up with, we do not expect the formalization to be significantly more difficult although it should be significantly longer.

**Cantor-Bernstein-Schröder.** As noted in [3], the construction of the division by two is closely related to the Cantor-Bernstein-Schröder (CBS) theorem. We recall that it states that given two sets $A$ and $B$ equipped with injections $A \hookrightarrow B$ and $B \hookrightarrow A$, there is a bijection $A \simeq B$. The proof can be performed in classical set theory without resorting to the axiom of choice. It has been known for some time that classical logic is necessary here: the theorem holds in a topos with a natural number object if and only if the topos is boolean [8, lemma D.4.1.12]. More recently, it has been shown that CBS is actually equivalent to the excluded middle (the new part being of course the left-to-right implication) [14]. Also recently, the CBS theorem has been generalized in the setting of homotopy type theory, it has been shown by Escardó (and also formalized in Agda) that any two *types* $A$ and $B$ equipped with mutual *embeddings* (which suitably generalizes the notion of injection) are *equivalent* [6]. Similarly, here, we show that division by two generalizes from sets to arbitrary types.

As for comparison, the situation regarding division by two is less explored. A non-boolean topos in which division by two cannot be performed was exhibited [17], showing that excluded middle is necessary to carry on the proof. However, we are not aware of an explicit internal proof that division by two implies excluded middle, so that we leave it as an interesting open question.

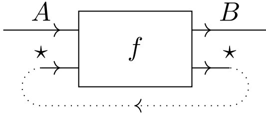
**Plan of the paper.** We first explain the baby case of subtraction by 1 in Section 2, recall Conway and Doyle's construction of division by 2 in Section 3, and the basics of homotopy type theory in Section 4. Our formalization is detailed in Section 5, and we explain the generalization to arbitrary types in Section 6.

## 2 Subtraction by 1

As a baby version of division by two, let us first present subtraction by one. The theorem we are aiming at proving is the following one:

▶ **Theorem 1.** *In intuitionistic ZF set theory (without choice), given two sets $A$ and $B$, if there is an isomorphism $A \sqcup 1 \simeq B \sqcup 1$, then there is an isomorphism $A \simeq B$.*

**Proof.** We denote by $\star$ the unique element of 1. Writing $f : A \sqcup 1 \leftrightarrows B \sqcup 1 : g$ for the two components of the isomorphism, we need to construct an isomorphism $f' : A \leftrightarrows B : g'$. We define $f'(a) = f(a)$ if $f(a)$ belongs to $B$ and $f'(a) = f(\star)$ otherwise (necessarily $f(\star)$ belongs to $B$ since, otherwise, we would have $f(a) = \star = f(\star)$ and $f$ would fail to be injective). Graphically, the construction of $f'$ from $f$ can be represented as below, which should be familiar to people knowledgeable about traced monoidal categories. The function $g'$ can be defined similarly, and the two can be checked to be inverse of each other since $f$ and $g$ are. ◀



The above proof can easily be formalized in Agda [13, `Sub1.agda`], let us detail it a bit as an illustration. We first define an operation which to an injective function $f : A \sqcup 1 \to B \sqcup 1$ associates its "restriction" $f' : A \to B$ as defined in the above proof. Naively, we are tempted to define $f'(a)$ by case analysis (i.e. pattern matching) on $f(a)$ and then by case analysis on $f(\star)$ when $f(a) = \star$. However, we cannot conclude by injectivity when $f(a) = f(\star)$ because of the way pattern matching works in Agda: when matching on $f(a)$, all occurrences of $f(a)$ are replaced by its value, but we do not keep the equality between $f(a)$ and its value, which we need here. The trick to overcome this, consists in matching not on $f(a)$ directly but on the singleton $f(a)$, where the *singleton* of an element `a` of type `A` is `singl a = Σ[ x ∈ A ] (a ≡ x)`, the type of pairs consisting of an element `x` of `A` together with an equality `a ≡ x` (we write `toSingl a` for the element `a` trivially seen as an element of this type). The restriction operation is thus defined as

```
restrict : {A B : Type} (f : A ⊔ ⊤ → B ⊔ ⊤) → isInjection f → A → B
restrict f inj a with toSingl (f (inl a))
... | inl b  , p = b
... | inr tt , p with toSingl (f (inr tt))
... | inl b  , q = b
... | inr tt , q = ⊥.rec (inl≢inr (inj (p · sym q)))
```

where `inl` and `inr` are the canonical injections in the coproduct and ⊤ is the type with one element `tt`. In the last case, we combine the equalities `p : f (inl a) ≡ inr tt` and `q : f (inr tt) ≡ inr tt` in order to obtain an equality `f (inl a) ≡ f (inr tt)`, from which we deduce by injectivity (the argument `inj`) that `inl a ≡ inr tt` which is impossible since the two components of a coproduct are disjoint (lemma `inl≢inr`). Finally, we can construct the "predecessor" we were looking for, by applying twice the above restriction function in order to construct the components of the isomorphism, and showing that they are mutually inverse (this requires reasoning by case analysis and using the same singleton trick as above):

```
predecessor : {A B : Type} → A ⊔ ⊤ ≃ B ⊔ ⊤ → A ≃ B
```

What have we gained by performing the formalization? We are now sure that it is entirely formal and that it does not use excluded-middle, since Agda works in intuitionistic Martin-Löf type theory (looking at the proof of Theorem 1 it is not immediately obvious that we are not using reasoning by contraposition in an essential way for instance). In a similar way, we can observe that this proof is still valid in the setting of homotopy type theory (or, more generally, without assuming axiom K). We thus obtain a generalization of Theorem 1: the operation $-\sqcup 1$ is not only regular for sets, but also for *spaces* (i.e. interpretations of types in homotopy type theory [9]). It also has some interesting consequences from the point of view of type theory. For instance, given a natural number $n$, we write `Fin` $n$ for the canonical set $\{0, 1, \ldots, n-1\}$ with $n$ elements. Formally, it is defined as the type `Fin n = Σ[ k ∈ ℕ ] (k < n)` of natural numbers strictly below $n$. It is easy to construct an isomorphism `Fin (suc n) ≃ Fin n ⊔ ⊤` between the canonical set with $n+1$ elements and the canonical set with $n$ elements with one element added. It is then easy to deduce by induction (on `m` and `n`) that the type constructor `Fin` is injective, in the sense that `Fin m ≃ Fin n` implies `m ≡ n`. Note that the equivalence in the argument can be replaced by an equality if we furthermore assume univalence (this fact can also be proved without univalence, but the known proofs are much more involved [10]).

## 3 The Conway-Doyle-Sierpiński construction of division by 2

Let us first describe briefly and informally the way division by 2 can be performed in classical ZF set theory without choice. We are mostly following the exposition given in [3], because of the nice "geometric" interpretation given there, but the construction is essentially quite similar to the predating one [16]. Suppose given two sets $A$ and $B$ and a bijection

$$f : A \times 2 \leftrightarrows B \times 2 : g \tag{1}$$

where $2 = \{-,+\}$ is a set with two elements. Our goal here is to perform division by 2, i.e. construct from this a bijection $A \simeq B$.

**The graph of a bijection.** The data (1) can be interpreted as a directed graph whose vertices are the elements of a quotient of $(A \times 2) \sqcup (B \times 2)$ and whose edges are the elements of $A \sqcup B$. Namely, we see an element $a \in A$ as an arrow with $(a,-)$ as source and $(a,+)$ as target, and similarly for $B$. We quotient the set of vertices and identify any two vertices which are related by the above bijection. For instance, with $A = \{a, a'\}$, $B = \{b, b'\}$, $f(a-) = b'-$, $f(a+) = b+$, $f(a'-) = b'+$ and $f(a'+) = b-$, the graph we construct is

$$a- = b'- \;\cdot\; \xrightarrow{a} \;\cdot_{\,a+=b+}\; \xleftarrow{b} \;\cdot\; b- = a'+ \qquad a- = b'- \;\cdot\; \xrightarrow{b'} \;\cdot_{\,a'-=b'+}\; \xrightarrow{a'} \;\cdot\; b- = a'+ \tag{2}$$

It can be noted that, in such a graph, every (undirected) path alternates between edges in $A$ (drawn with plain lines) and $B$ (drawn with dashed lines), and that any vertex is incident to exactly two edges (and this exactly characterizes the graphs constructed in this way).

**Chains.** We can partition the edges of the graph into connected components, that we call *chains*: any two edges in the same chain are related by a non-directed path. In order to construct the bijection $A \simeq B$, it is clearly enough to construct, for each chain, a bijection between the elements in $A$ and those in $B$. It can be observed that, given a chain, if we fix an *origin* edge $e$ in this chain, we have a canonical way of constructing such a bijection: every edge in the chain is reachable by a non-directed path, and we send each edge in $A$ to the edge in $B$ just "after" (according to this path) and each edge in $B$ to the edge just "before" (according to this path). In other words, our bijection *swaps* each element of $A$ (resp. $B$) with the next (resp. previous) one. Note that in order to make sense of the the notion of previous/next element in a chain, we need to fix a global orientation on the chain, which is canonically done by fixing the origin edge $e$. For instance, if we take $a$ as origin in (2), the edge $a'$ is reached by the path $aba'$ and therefore the swapping bijection sends it to the "next" edge, which is $b'$ and dually $b'$ is sent to $a'$ (and similarly, the bijection exchanges $a$ and $b$). However, if we had taken $b$ as origin, the bijection would have swapped $a$ with $b'$ and $a'$ with $b$.

**Constructing the bijection.** By the previous discussion, all we are left to do in order to construct a bijection between $A$ and $B$ is to pick an element in each chain. However, we do not have any immediate way of performing this since we are not accepting the use of the axiom of choice here. Note that this does not mean that we cannot perform this, only that we are not allowing to do this "by magic": in order to exhibit an element, we must construct it explicitly. The insight of Conway and Doyle to do so is the following one. In a given undirected path, we

can interpret an edge taken forward as an "opening bracket" and an edge taken backward as a "closing bracket" (for some reason, this is the inverse of the convention taken in [3]). We say that a path is *well-bracketed*, when the sequence of brackets it induces is, in the usual sense. We say that an edge $e$ is *matched* when there is a path starting with $e$ as opening bracket, which is well-bracketed: the edge closing the bracket corresponding to the first edge is called the *matching edge*. For instance, in (2), the edge $a$ is matched because the path $ab$ is well-bracketed (it corresponds to the sequence "()" of brackets), but the edge $b'$ is not: for instance, the path $b'a'ba$ is not well-bracketed since it corresponds to the sequence "(()(". It is not difficult to see that that for a matched edge in $A$, the matching edge is always in $B$, and conversely. Consider a given chain, we have three cases

- if every edge is matched then there is no obvious way to pick a particular one, but we can send each edge to the matching one, and this provides a bijection between the elements in $A$ and the elements in $B$ of the chain,
- otherwise, if we remove all matched edges then the chain must be of one of the following forms:

$$\underset{\text{(a)}}{\cdots\cdot\to\cdot\to\cdot\to\cdot\to\cdot\cdots} \qquad \underset{\text{(b)}}{\cdots\cdot\leftarrow\cdot\leftarrow\cdot\overset{e}{\leftarrow}\cdot\overset{e'}{\to}\cdot\to\cdot\to\cdot\cdots} \qquad \underset{\text{(c)}}{\cdots\cdot\leftarrow\cdot\leftarrow\cdot\leftarrow\cdot\leftarrow\cdot\cdots} \tag{3}$$

  since once convinces himself that it would otherwise contain a matched edge
  - in the case (b), the edges $e$ and $e'$ are called *switching edges*, one of them is in $A$: we can take it as origin, and the associated swapping bijection as described above provides a bijection between the elements in $A$ and those in $B$ in the chain,
  - in the cases (a) and (c), all the edges are oriented in the same direction and we can take the swapping bijection associated to any of those (the bijection will not depend on the choice of the origin).

While the above reasoning can reasonably be considered as a proof, the reader should note that there are many points which are not entirely precise. For instance, when an edge is reachable from another there might be multiple paths between them (for instance, when the graph has loops) and we should make sure that our reasoning does not depend on the choice of a path. Also, in the above drawings (3), we have not exactly drawn the possible chains but the possible maximal paths in the chain, since the chain itself might have loops in the cases (a) and (c). Also, when we consider edges above we actually often implicitly consider them traveled in a particular direction. Also, in the last case, it is a bit puzzling that we cannot pick an edge on a chain (because we are not accepting the axiom of choice), but we can perform a construction using an edge of the chain as long as the result does not actually depend on the choice of this edge. The formal developments performed here should hopefully clarify all those points (and more).

## 4 A primer in homotopy type theory

In this section, we make a brief reminder of the concepts in homotopy type theory that we are going to use and refer the reader to the reference book [19] for details: in a sentence, we work in a variant of Martin-Löf type theory validating the univalence axiom and supporting higher inductive types. The precise formalization of it we use here is the one provided by the cubical variant of the Agda proof assistant and the associated library [20].

**Equality.** We write `Type` for the universe of small types and call *types* its elements (for simplicity, we do not explicitly deal with universe levels here). Given a type `A` and two terms $x$ and $y$ of this type, we write $x \equiv y$ for the type of *equalities* (or *identities* or *paths*) between them: this relation can internally be shown to be an equivalence relation. A type is a *proposition* when any two of its elements are equal, i.e. it satisfies the predicate `isProp A` defined as `(x y : A) → x ≡ y`. Similarly, a type is a *set* when any two paths between any two elements are equal (otherwise said, the type $x \equiv y$ is a proposition for any two terms $x$ and $y$ of this type). One of the main properties of equality is

```
transport : {A B : Type} → A ≡ B → A → B
```

which expresses that when two types `A` and `B` are equal any element of the first can be seen as an element of the second.

**Equivalences.** A map `f : A → B` is an *equivalence* when there exists a map `g : B → A` such that both `g ∘ f` and `f ∘ g` are identities (for subtle reasons, the actual definition of equivalence actually has to be slightly different from this [19, chapter 4], but this will play no role here). Given such a map, we say that the types `A` and `B` are *equivalent*, what we write `A ≃ B`. Given two types `A` and `B`, there is a canonical map `A ≡ B → A ≃ B` and the *univalence* axiom states that this map is itself an equivalence: homotopy type theory (HoTT) postulates this axiom. One can construct a model of HoTT where types are interpreted not as booleans or sets, but as *spaces* [9].

**The axiom of choice.** One has to be careful when postulating non-constructive principles in HoTT. Traditionally, classical logic is defined as validating the excluded-middle `A ∨ ¬ A` for every type `A`. Postulating this is inconsistent with the univalence axiom [19, section 3.4], however it is consistent to postulate the excluded middle for every proposition (as opposed to type) `A`, which is what we mean here by *classical logic*. Also traditionally, in set theory, the axiom of choice states that every family of non-empty sets it non-empty. In the appropriate type theoretic formulation of this, rather than saying that a set `A` is non-empty, we want to express that "we know that there exists an element of `A`", which corresponds to the type `∥ A ∥`, called the propositional truncation of `A` (see below). The formulation of the axiom of choice is thus [19, section 3.8]:

```
(A : Type) (f : A → Type) → ((x : A) → ∥ f x ∥) → ∥ ((x : A) → f x) ∥
```

It is known that both axioms and their negations are consistent with univalence.

**Higher inductive types.** It is common for functional languages to feature inductive types, whose elements are freely generated by constructors: typically, natural numbers are generated by zero and successor. Cutting-edge implementations of HoTT, such as the cubical variant of Agda, support the more general *higher inductive types* [19, chapter 6] which allow, in addition to traditional constructors, constructors for equalities between the elements of the type. For instance, the *propositional truncation* operation ∥_∥ mentioned above can be defined by

```
data ∥_∥ (A : Type) : Type where
  |_|    : A → ∥ A ∥
  squash : (x y : ∥ A ∥) → x ≡ y
```

which indicates that is has one traditional constructor |_| allowing to see any element of `A` as an element of `∥ A ∥`, and a constructor `squash` which adds an equality between any two elements of `∥ A ∥`: thanks to this last constructor, `∥ A ∥` can always be shown to be a proposition.

The elimination principle states that any function `A → B`, where `B` is a proposition, induces a function ∥ `A` ∥ `→ B`. One can similarly define the *set truncation* ∥ `A` $\|_0$ of a type which produces a set from `A` in a universal way. The elimination principle states that a function `A → B` sending elements in relation to equal ones and where `B` is a set induces a function ∥ `A` $\|_0$ `→ B`. Another typical construction which can be defined as a higher inductive type is the quotient set `A / R` of a type `A` by a relation `R`, of type `A → A → Type` (this construction internally uses set truncation in order to produce a set).

## 5 Implementation in Agda

We now present our formalization in cubical Agda of the division algorithm described in previous section for sets (and generalize it to arbitrary types in next section). The interested reader can access the code on the repository [13], which should be compatible with Agda 2.6.1 and the version 0.2 of the cubical library, and takes more than 3000 lines of code. We do not detail the syntax of Agda, but it should hopefully be sufficiently close to the usual mathematical notations to be readable by a non-expert. For full disclosure, the proof is mostly complete apart from a few minor points: the integer module of the standard library is not very complete and we postulated most of the standard properties (e.g. the group structure), a few simple combinatorial lemmas were not shown due to lack of time (they always come with a detailed explanation and should be completed soon, but the lack of automation in Agda sometimes make simple properties long to show). Also, some parts of the proof in the definition of the swapping bijection cannot be checked in reasonable time: we have a hole whose type indicates a property to be shown, we have a lemma which shows this exact property, but putting the lemma into the hole makes the typechecker loop. This can most likely be fixed by preventing the reduction in some parts of the proofs (we have successfully done this in other places, by using the `abstract` keyword). The main result is a constructive proof of the following theorem in HoTT with excluded-middle (without supposing the axiom of choice), where 𝟚 is a type with two elements (e.g. the booleans):

▶ **Theorem 2.** *Given two types* `A` *and* `B` *which are sets and an equivalence* `A × 𝟚 ≃ B × 𝟚`*, we have an equivalence* `A ≃ B`.

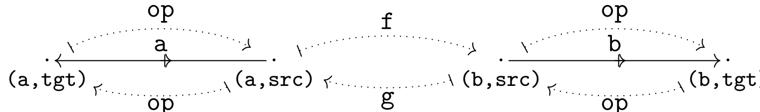
**Arrows.** In the following, we fix two sets `A` and `B` and the equivalence `A × 𝟚 ≃ B × 𝟚`, whose components are denoted `f : A × 𝟚 → B × 𝟚` and `g : B × 𝟚 → A × 𝟚`. We write `src` and `tgt` for the two elements of 𝟚, because they are thought of as indicating the *end* of an arrow: either source or target. Following the description of the data as a graph given in Section 3, we define the type of *arrows* as `Arrows = A ⊔ B`: an arrow is either an element of `A` or `B` and we define its *polarity* to be negative or positive accordingly. Moreover, the type of *ends* is `Ends = Arrows × 𝟚` (this is the collection of all ends of all our arrows), see [13, `Arrows.agda`]. We thus think of an element `a` of `Arrows` as on the left:

(a , src) ·—a→· (a , tgt)    ∗—fw a→·    ·—bw a→∗

All the arrows we consider are directed (in the sense that they have a source and a target) and we keep the terminology of *directed arrow* for an arrow equipped with a traveling direction, which is either *forward* or *backward* (we sometimes speak of a *non-directed* arrow for an arrow without a choice of direction). In practice, it is convenient to encode the direction of an arrow by its starting end, so that we define the type of directed arrows as `dArrows = Ends`, with the convention that `(a , src)` (resp. `(a , tgt)`) is the arrow `a` traveled forward (resp. backward)

and write `fw` (resp. `bw`) for the function of type `Arrows → dArrows` orienting an arrow forward (resp. backward), as indicated on the above picture (the starred end is the one which is used to identify the direction, and the triangle in the middle of the arrow represents the direction). We also write `arrow : dArrows → Arrows` and `end : dArrows → 𝟚` for the two projections, respectively associating to a directed arrow its underlying arrow and end. The discussion of previous section, should have convinced you that the whole difficulty of dividing by 2 lies in the ability of determining a consistent orientation of the arrows in each chain.

**Reachability.** We denote by `op : dArrows → dArrows` the function which reverses the orientation of a directed arrow (by swapping `src` and `tgt` in the second component). We can then define the function `next : dArrows → dArrows` that associates to each directed arrow the next directed arrow when traveling it in the chosen direction. For instance, suppose given an arrow `a` in `A` taken backward: the next arrow `b` can be computed by first applying `op` (in order to "travel across `a`") and then apply `f` in order to obtain the end which corresponds to the next directed arrow, which is necessarily in `B`:




Other cases are handled similarly, and we can dually define a `prev` function which computes the previous arrow. From previous functions, by induction we can define function

```
iterate : ℤ → dArrows → dArrows
```

which gives the arrow reached after traveling for $n$ steps (with the convention that we travel in the opposite direction of the arrow when $n$ is negative). This defines an action of $\mathbb{Z}$ on directed arrows, in the sense that `iterate 0` is the identity, `iterate (m + n) ≡ iterate n ∘ (iterate m)` and `iterate n ∘ op ≡ op ∘ (iterate (- n))` for `m` and `n` integers. It can also be shown that it is *alternating*, in the sense that `iterate n` preserves the polarity when `n` is even and inverts it when it is odd.

Iteration allows us to define a reachability relation on directed arrows as follows:

```
reachable : dArrows → dArrows → Type
reachable e e' = Σ[ n ∈ ℤ ] (iterate n e ≡ e')
```

Two arrows `e` and `e'` are reachable when the second can be obtained by iterating from the first. Note that a proof of `reachable e e'` is the *data* of an integer representing the number of steps between `e` and `e'` (which is essentially the same as giving a path between `e` and `e'` in the graph since every vertex has exactly two neighboring edges), together with a proof that this is the case. It is also useful to consider the following variant defined by

```
is-reachable e e' = ∥ reachable e e' ∥
```

which is closer to the reachability in the usual sense: it asserts the existence of a path between `e` and `e'`, without a priori providing a particular one. One can also define a variant, which expresses the reachability of arrows (in `Arrows`) with the relation

```
reachable-arr a b = Σ[ n ∈ ℤ ] (arrow (iterate n (fw a)) ≡ b)
```

which expresses that when traveling from the arrow `a` (oriented in the forward direction, but this plays little role since we can travel forward [when `n` is positive] or backward [when `n` is negative]) one can reach a directed arrow which is `b` (with some direction). And of course, one can similarly define a relation `is-reachable-arr` by propositional truncation. Those relations can easily be shown to equivalence relations.

**Revealing reachability.** Given any two directed arrows `e` and `e'`, it is easy to show the implication `reachable e e' → is-reachable e e'`: if we are provided with a path between `e` and `e'` then there exists one between `e` and `e'`. What is perhaps more surprising is that the converse implication holds, i.e. if there exists a path between the arrows, we can always construct it (we like to think that this operation *reveals* the path):

▶ **Proposition 3.** *Given directed arrows* `e` *and* `e'`*,* `is-reachable e e'` *implies* `reachable e e'`*.*

**Proof.** It is folklore that ℕ is *searchable* (see [5] and [13, `Nat.agda`]): given a predicate `P : ℕ → Type` such that `P n` is a decidable proposition for every natural number `n`, we have that ‖ `Σ ℕ P` ‖ implies `Σ ℕ P` (if there is a natural number satisfying `P` then we can construct it). Namely, one can consider the predicate `Q` on natural numbers defined by

```
Q n = P n × ((m : ℕ) → P m → n ≤ m)
```

which expresses that `n` is a smallest natural number satisfying `P`. Since ℕ is a set and the order is total, the type `Σ ℕ Q` is a proposition. We can therefore eliminate ‖ `Σ ℕ P` ‖ to it, from which we can easily deduce an element of `Σ ℕ P` by projection.

Since ℤ and ℕ are isomorphic (for instance, via the map sending $n$ to $2n$ or $-2n+1$ depending on whether $n$ is positive or negative), they are equal by univalence, and ℤ is also searchable. All we are left to show is that, for every integer `n`, the property `P n = iterate n e ≡ e'` is a decidable proposition. Since `A` and `B` are supposed to be sets, the type `dArrows = (A ⊔ B) × 2` is also a set and thus `P n` is a proposition (as an equality between two elements of a set). It is moreover decidable because we assume the excluded middle. ◀

Note that the proof uses both our main hypothesis: that `A` and `B` are sets and that the excluded-middle holds. A similar property of course holds for `reachable-arr`.

**Orientation.** We have already noted that fixing an edge $e_0$ induces an orientation for every reachable arrow $e$, namely the traveling direction when reaching the edge $e$ from $e_0$. Another important property is that this orientation is well-defined, in the sense that it does not depend on the actual path from $e_0$ to $e$:

▶ **Proposition 4.** *Given directed arrows* $\texttt{e}_0$*,* `e` *and* `e'`*, such that* `reachable` $\texttt{e}_0$ `e` *and* `reachable` $\texttt{e}_0$ `e'` *and* `arrow e ≡ arrow e'`*, we have* `end e ≡ end e'`*.*

**Proof.** By symmetry and transitivity of reachability, we know that `e'` is reachable from `e` and we reason by induction on the length $n$ of the path from `e` to `e'`. If $n = 0$, the result is immediate. The case $n = 1$ is impossible because paths are alternating and `e` and `e'` have the same polarity because they have the same underlying arrow. Otherwise, we reason by case analysis on the respective ends of `e` and `e'`: if they are the same then we are done. Otherwise, the path is of the form `e` · $\texttt{e}_1$ · ⋯ · $\texttt{e'}_1$ · `e'` and we can apply the induction hypothesis to the path $\texttt{e}_1$ · ⋯ · $\texttt{e'}_1$ and from which we deduce the result. ◀

**Chains.** Our goal is now to give a type which describes chains. The chain of a directed arrow `e` is the set of directed arrows which are reachable from it. Naively, this would suggest defining the type of chains as

```
Σ[ e ∈ dArrows ] (Σ[ e' ∈ dArrows ] (is-reachable e e'))
```

i.e. the sets of arrows `e'` which are reachable from some arrow `e`. However, this type is rather the type of *pointed* chains, i.e. the type of chains together with a distinguished arrow, and we have explained in the introduction that this choice of distinguished point is precisely the crux of our construction.

Fortunately, we have access to quotient types and we can define the *directed chains* as the quotient of arrows under the reachability predicate:

```
dChains = dArrows / is-reachable
```

As a side note, although the above definition is slightly more convenient (see below), we would have obtained an equivalent type if we had defined chains as `dArrows / reachable`: this is a general fact that quotienting a type under a relation or under the propositional truncation of the relation give equivalent types. There is a non-directed analogous definition for (non-directed) arrows, and we define the type of (non-directed) chains as:

```
Chains = Arrows / is-reachable-arr
```

The function `delements : dChains → Type`, which associates to a directed chain its *elements*, i.e. the directed arrows in the equivalence class, can be defined as `delements c = fiber [_] c`, where `fiber f y = Σ[ x ∈ A ] (f x ≡ y)` associates to a function `f` and an element `y` its homotopy fiber (the type of preimages of `y` under the function) and `[_] : dArrows → dChains` is the quotient map (a similar function `elements` can be defined for non-directed chains).

▶ **Proposition 5.** *Any two arrows in the same chain are reachable one from the other.*

**Proof.** The relation `is-reachable-arr` being proposition-valued (since it is defined by propositional truncation), it is *effective*, which means any two directed arrows in the same directed chain are related by `is-reachable-arr`, and thus by `reachable-arr` thanks to Proposition 3. ◀

Similar properties hold in the directed variant, but we will use the above proposition.

**Pointed chains.** Given a directed arrow `o` (for "origin"), we think of the directed chain `[ o ]`, i.e. its equivalence class, as being the pointed chain associated to `o`. It is not difficult to construct a map

```
delements [ o ] → elements [ arrow o ]
```

which takes a directed arrow reachable from `o` to the underlying arrow, which is reachable from the underlying arrow of `o`. More, interestingly, for every (non-directed) arrow `o`, there is map

```
elements [ o ] → delements [ fw o ]
```

Namely, given an element of `elements [ o ]`, i.e. an arrow `a` such that `[ o ] ≡ [ a ]`, there is an integer `n` such that `a` can be obtained from `o` by iterating `n` times, and we define the image as the directed arrow obtained by iterating `n` times from `fw o`. By using Proposition 4, one can show that these maps form an equivalence, thus showing that picking an arrow in a non-directed chain equips it with a canonical orientation:

▶ **Proposition 6.** *Given a non-directed arrow* `o`*, there is an equivalence*

```
elements [ o ] ≃ delements [ fw o ]
```

**Well-bracketed chains.** Suppose given a directed arrow `e`. Given an integer `n`, the *height* of the path of length `n` is the sum for `i` between `0` (included) and `n` (excluded) of the *weight* of the directed arrow obtained by iterating `i` times from `e`, this weight being 1 (resp. −1) if it is in the forward (resp. backward) direction. For instance, the following path of length 4 has height 2:

$$\cdot \xrightarrow{1} \cdot \xrightarrow{1} \cdot \xleftarrow{-1} \cdot \xrightarrow{1} \cdot$$

We say that a (non-directed) `a` is *matched* when there is a positive integer `n` such that the directed arrow `e` obtained by iterating `n` times from `fw a` (the *matching arrow*) is at height 0 and all intermediate arrows have strictly positive height:

```
matched a = Σ[ n ∈ ℕ ] (height (suc n) (fw a) ≡ 0 ∧
                        ((k : ℕ) → k < suc n → ¬ (height k (fw x) ≡ 0)))
```

The chain of a (non-directed) arrow `o` is *well-bracketed* when every arrow `a` reachable from `o` is matched.

▶ **Proposition 7.** *Being well-bracketed for a reachable arrow is a proposition, which is independent of the choice of* `o`*.*

**Proof.** Being a proposition follows from the fact that any two matching of a given arrow are necessarily equal, which follows from the definition of matching. Independence of the origin follows from the fact that any two possible origins are reachable from the other by Proposition 5. ◀

Given a non-directed chain `c`, we finally say that it is *well-bracketed* when its elements are: reading an arrow in the forward (resp. backward) direction as an opening (resp. closing) bracket, this amounts to say that the resulting word is well-bracketed in the traditional acceptation. In order for this definition to make sense, we need to eliminate to a set (because quotient is defined by set truncation): here, we eliminate to the type of propositions (also called HProp) which is known to be a set, of which being well-bracketed is an element by Proposition 7. In order to use the elimination principle, we must show that the result does not depend on the choice of the element in the equivalence class, which is precisely the second part of Proposition 7 (other properties on chains below are defined in a similar way, even though we do not detail this).

▶ **Proposition 8.** *Given a well-bracketed chain* `c`*, we have an equivalence* `chainA c ≃ chainB c`*.*

**Proof.** The function sending an arrow to the matching one (which exists because the chain is well-bracketed and is unique by Proposition 7) can be shown to be involutive and swaps polarity because a matching arrow is necessarily at even distance from the original arrow. As in the previous definition, we must show that the type `chainA c ≃ chainB c` is a set (which is the case essentially because `A` and `B` are sets) and that the definition does not depend on the choice of the element. ◀

**Switching chain.** A (non-directed) arrow a is *switch* when it is not matched, and the next non-matched arrow (going in the backward direction indicated by the arrow) is in the opposite direction. For instance, the arrow $a$ below is switching because the next arrow which is not bracketed, namely $b$, is in the opposite direction:

$$\ldots \xleftarrow{a} \cdot \xrightarrow{(} \cdot \xleftarrow{)} \cdot \xrightarrow{b} \ldots$$

Formally, this can be defined as follows [13, Switch.agda]:

```
switch a = ¬ (matched a) ∧ Σ[ n ∈ ℕ ] (
   let b = iterate (fromℕ n) (bw a) in
   (end b ≡ src) ∧ ¬ (matched (arrow b)) ∧
   ((k : ℕ) → suc k < n → matched (arrow (iterate (fromℕ (suc k)) (bw a)))))
```

It is easy to show that being switch for an arrow is a proposition. Finally, we say that a chain is switching when one of its elements is a switch arrow in $A$.

▶ **Proposition 9.** *The property of being switching for a chain is a proposition.*

**Proof.** This amounts to show that there is at most one switch arrow in a chain. First note that it is important that we require that the switch arrow we are looking for is in $A$ (otherwise, the associated arrow, which can be shown to be in $B$, would also be switch). If there were two switch arrows, by case analysis on their relative positions, one of them can be shown to be matched, thus contradicting the definition. ◀

**Slopes.** We say that a directed arrow o is *sequential* when any two directed arrows which are reachable from o and not matched are oriented in the same direction. Formally,

```
sequential o = ((m n : ℤ) →
    let a = iterate m o in
    let b = iterate n o in
    ¬ (matched (arrow a)) → ¬ (matched (arrow b)) → end a ≡ end b)
```

This definition can be shown to be independent from the choice of o in a chain and from its direction, so that we can define the notion of *sequential* (non-directed) chain. Finally, we say that a chain c is a *slope* when it is sequential and there exists one of its elements which is not matched, in the sense that we have

```
‖ Σ[ a ∈ elements o ] ¬ (matched a) ‖        (4)
```

▶ **Proposition 10.** *The property of being a slope for a chain is a proposition.*

**The trichotomy.** Because being well-bracketed and being switching are propositions for a chain, we can use the excluded middle on those. Moreover, it can be shown that a chain which is not switching is sequential. From there, we easily deduce the following principle of "trichotomy" [13, Tricho.agda]:

▶ **Proposition 11.** *Any (non-directed) chain is either well-bracketed, switching or sequential.*

**Swappers.** Given a non-directed chain `c`, we write `chainA c` (resp. `chainB c`) for the elements in `A` (resp. `B`) of the chain. We use similarly the notations `dchainA c` and `dchainB c` for the elements of a directed chain `c`. Since the elements in a chain a canonically oriented by a choice of the origin (Proposition 6), we have the following relationship, for every (non-directed) arrow `o`:

```
chainA [ o ] ≃ dchainA [ fw o ]
```

(and similarly for the component `B`). Given a directed arrow `o`, one can construct an equivalence

```
dchainA [ o ] ≃ dchainB [ o ]
```

by sending each element in `A` (resp. `B`) of the chain `[ o ]` to the next (resp. previous) element. By the above, for every non-directed arrow `o`, we thus have an equivalence

```
chainA [ o ] ≃ dchainA [ fw o ] ≃ dchainB [ fw o ] ≃ chainB [ o ]
```

This equivalence can be shown to be independent of the choice of `o` in its reachability class, and this thus induces a function

```
(c : Chains) → elements c → chainA c ≃ chainB c        (5)
```

Moreover, when we have a chain which is a slope, we can also build such a bijection

```
(c : Chains) → slope c → chainA c ≃ chainB c        (6)
```

In order to construct it, we essentially need to show that the bijection (5) does not depend on the choice of the non-matched element of the chain `c` in order to eliminate the propositional truncation (4).

**Chainwise bijection.** We are now in position of proving our main theorem. We first observe that we can build the equivalence we are looking for "locally", by which we mean "chain by chain", in the following sense:

▶ **Proposition 12.** *If, for every chain* `c` *we have* `chainA c ≃ chainB c`*, then* `A ≃ B`*.*

**Proof.** Given a relation `R` on a type `A`, the type is the union of its equivalence classes in the sense that we have `A ≃ Σ[ c ∈ A / R ] (fiber [_] c)`. The result can be deduced from this and standard equivalences. ◀

▶ **Theorem 2.** *Given two types* `A` *and* `B` *which are sets and an equivalence* `A × 𝟚 ≃ B × 𝟚`*, we have an equivalence* `A ≃ B`*.*

**Proof.** By Proposition 12 is it enough to construct an equivalence `chainA c ≃ chainB c` for any chain `c`. By Proposition 11, such a chain is either well-bracketed, in which case we conclude by (8), or swapping, in which case we conclude by (5) applied to the swapping arrow, or slope, in which case we conclude by (6). ◀

## 6 Generalization to arbitrary types

We bring here our main novel contribution by showing that division extends to arbitrary types (as opposed to sets), whose main arguments are formalized in [13, `Spaces.agda`]. The proof is based on Theorem 2 and the following observation.

Given a type `A`, we write $\|$ `A` $\|_0$ for its set truncation and $|_|_0$ `:` `A` `→` $\|$ `A` $\|_0$ for the quotient map. Given an element `a` of `A`, we think of $|$ `a` $|_0$ as the *connected component* of `a` and `fiber` $|_|_0$ $|$ `a` $|_0$ as the elements of this connected component, which can be justified by the fact that this type is equivalent to `Σ[ a' ∈ A ]` $\|$ `a ≡ a'` $\|$, i.e. the elements of `A` for which there exists a path to `a`. The following two propositions will allows us to work with equivalences connected component by connected components, i.e. fiberwise with respect to $|_|_0$:

▶ **Proposition 13.** *An equivalence* `e : A ≃ B` *with underlying function* `f : A → B` *induces, for every connected component* `x :` $\|$ `A` $\|_0$*, an equivalence*

`fiber` $|_|_0$ `x ≃ fiber` $|_|_0$ `(`$\|\|_0$`-map f x)`

▶ **Proposition 14.** *Given an equivalence* `e : A ≃ B` *with underlying function* `f : A → B`*, and type families* `P : A → Type` *and* `Q : B → Type`*, which are pointwise equivalent, in the sense that* `P x ≃ Q (f x)` *for every* `x : A`*, the total spaces are equivalent:* `Σ A P ≃ Σ B Q`*.*

Now, suppose fixed two arbitrary types `A` and `B`. The type $\|$ `dArrows` $\|_0$ `=` $\|$ `(A ⊔ B) ×` $\mathbb{2}$ $\|_0$ of connected components of directed arrows and the type `(`$\|$ `A` $\|_0$ `⊔` $\|$ `B` $\|_0$`) ×` $\mathbb{2}$ of arrows in connected components are canonically equivalent (and we implicitly identify the two here) because set truncation commutes with disjoint union and products with sets.

▶ **Proposition 15.** *Two directed arrows* `a` *and* `b` *in* $\|$ `dArrows` $\|_0$ *which are reachable from one another have the same connected components:* `fiber` $|_|_0$ `a ≃ fiber` $|_|_0$ `b`*.*

**Proof.** By recurrence on the length of path and symmetry, it is enough to show the result in the case where `b` is the next arrow after `a`. The function `next : dArrows → dArrows` is easily shown to be an equivalence (with `prev` as inverse) and thus induces an equivalence between `fiber` $|_|_0$ `a` and `fiber` $|_|_0$ `(`$\|$ `next` $\|_0$ `a)` by Proposition 13. ◀

▶ **Theorem 16.** *Given two types* `A` *and* `B` *and an equivalence* `A ×` $\mathbb{2}$ `≃ B ×` $\mathbb{2}$*, we have an equivalence* `A ≃ B`*.*

**Proof.** Suppose given two types `A` and `B` and a map `f : A ×` $\mathbb{2}$ `→ B ×` $\mathbb{2}$ which is an equivalence. By set truncation, it induces a map $\|$ `A ×` $\mathbb{2}$ $\|_0$ `→` $\|$ `B ×` $\mathbb{2}$ $\|_0$ and thus a map $\|$ `A` $\|_0$ `×` $\mathbb{2}$ `→` $\|$ `B` $\|_0$ `×` $\mathbb{2}$. We can apply Theorem 2 and deduce the existence of a map $f_0$ `:` $\|$ `A` $\|_0$ `→` $\|$ `B` $\|_0$ which is an equivalence. Because the way $f_0$ is constructed (it either sends parenthesis to matching ones or swaps an element of a chain with the next one) it can be shown that any element `a` of $\|$ `A` $\|_0$, seen as a (non-directed) arrow in $\|$ `A` $\|_0$ `⊔` $\|$ `B` $\|_0$, is sent to a reachable arrow `b`. This means that the corresponding directed arrow `fw a` is sent to either `fw b` or `bw b`, the second being reachable from the first, and thus that the `fiber` $|_|_0$ `a` and `fiber` $|_|_0$ `b` are equivalent by Proposition 15. We can conclude with the following series of equivalences:

`A ≃ Σ[ a ∈ A ] (fiber` $|_|_0$ `a) ≃ Σ[ b ∈ B ] (fiber` $|_|_0$ `b) ≃ B`

where the bijection in the middle follows by Proposition 14 from the fact that the types $\|$ `A` $\|_0$ and $\|$ `B` $\|_0$ are isomorphic (by Theorem 2) and have equivalent fibers under $|_|_0$ by the above reasoning. ◀

## 7 Conclusion and open questions

We have described our formalization of division by 2 of types in Agda, following the proof of Conway and Doyle. It fills in often left over details, such as bringing in the distinction between non-directed and direct arrows, the formalization of chains, the elimination of propositions and so on. It also allowed us to generalize the proof to arbitrary types.

**Practical lessons.** It also illustrates the usefulness of homotopy type theory, even when formalizing results about sets, most notably by bringing in quotient types and identity types with computational rules (and our development does rely quite a lot on the possibility of computing the result of transporting values along equalities, even though we did not insist so much on it in the paper). This work moreover shows that this is doable in practice: the library provided along with cubical Agda is rich enough (even though we mostly used v0.2 because of compatibility issues) and quite abstract (we almost never had to explicitly manipulate terms involving the cubical constructions, so that the proof is in principle quite independent of the precise formalization of HoTT in use). The first thing we observed is that not having definitional J rule (because it is incompatible with the cubical model) is sometimes quite cumbersome, although it is nothing compared to what has to be done in traditional Agda (in which J is definitional, but univalence does not compute, and HITs have to be axiomatized by hand). Another lesson we learned is that, in cubical Agda, it is much more manageable to use elimination rules associated to (higher) inductive types than directly use pattern matching: even though they are a priori less convenient, elimination principles avoid having to explicitly deal with cubical (interval) variables. A last thing we learned and already mentioned is that one should sometimes be careful to prevent the typechecker from computing some parts of the proof, in order not to make its computation time explode; more generally, one should be careful about implementing things in an efficient way (e.g. transporting an equivalence is sometimes out of reach whereas transporting the underlying function is).

**Future work.** As mentioned in the introduction, one can show that it is necessary to postulate the excluded-middle by using semantic arguments [17], but it would be interesting to have a constructive proof that division by two implies excluded-middle (in a similar fashion as for Cantor-Bernstein-Schröder [14]); we could unfortunately not find such a proof. Generalizations to natural numbers greater than two have been studied on paper [18, 3, 4] and could also be formalized in principle. Lastly, it would be interesting to investigate how division generalize to "numbers" which are not (finite) sets, in the sense that they have non-trivial higher-dimensional features.

## References


1 Felix Bernstein. Untersuchungen aus der Mengenlehre. *Mathematische Annalen*, 61(1):117–155, 1905.

2 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs*, number 69 in LIPIcs, page 262. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. arXiv:1611.02108.

3 John Conway and Peter Doyle. Division by three, 1994. arXiv:math/0605779.

4 Peter G Doyle and Cecil Qiu. Division by four, 2015. arXiv:1504.01402.

5 Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with Agda, 2019. arXiv:1911.00580.

**6** Martín Hötzel Escardó. The Cantor–Schröder–Bernstein Theorem for $\infty$-groupoids. *Journal of Homotopy and Related Structures*, 16(3):363–366, 2021. `arXiv:2002.07079`.

**7** Arie Hinkis. *Proofs of the Cantor-Bernstein theorem. A mathematical excursion*, volume 45 of *Science Networks Historical Studies*. Birkhäuser, 2013.

**8** Peter T Johnstone et al. *Sketches of an Elephant: A Topos Theory Compendium: Volume 2*, volume 2. Oxford University Press, 2002.

**9** Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. `arXiv:1211.2851`.

**10** Donnacha Oisín Kidney. A Small Proof that Fin is Injective. `https://doisinkidney.com/posts/2019-11-15-small-proof-fin-inj.html`, 2019.

**11** Adolf Lindenbaum and Alfred Tarski. *Communication sur les recherches de le théorie des ensembles*. 1926.

**12** Patrick Lutz. Conway Can Divide by Three, But I Can't, 2021.

**13** Samuel Mimram and Émile Oleon. Division by two in Agda, 2022. URL: `https://github.com/smimram/div2`.

**14** Pierre Pradic and Chad E Brown. Cantor-Bernstein implies Excluded Middle, 2019. `arXiv:1904.09193`.

**15** Rich Evan Schwartz. Pan galactic division. *The Mathematical intelligencer*, 37(3):8–10, 2015. `arXiv:1504.02179`.

**16** Wacław Sierpiński. Sur l'égalité $2m = 2n$ pour les nombres cardinaux. *Fundamenta Mathematicae*, 1(3):1–6, 1922.

**17** Andrew Swan. On Dividing by Two in Constructive Mathematics, 2018. `arXiv:1804.04490`.

**18** Alfred Tarski. Cancellation laws in the arithmetic of cardinals. *Fundamenta Mathematicae*, 36(1):77–92, 1949.

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

**20** Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31, 2021.

# Type-Based Termination for Futures

**Siva Somayyajula**
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

**Frank Pfenning**
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

## Abstract

In sequential functional languages, sized types enable termination checking of programs with complex patterns of recursion in the presence of mixed inductive-coinductive types. In this paper, we adapt sized types and their metatheory to the concurrent setting. We extend the semi-axiomatic sequent calculus, a subsuming paradigm for futures-based functional concurrency, and its underlying operational semantics with recursion and arithmetic refinements. The latter enables a new and highly general sized type scheme we call *sized type refinements*. As a widely applicable technical device, we type recursive programs with infinitely deep typing derivations that unfold all recursive calls. Then, we observe that certain such derivations can be made infinitely wide but finitely deep. The resulting trees serve as the induction target of our strong normalization result, which we develop via a novel logical relations argument.

**2012 ACM Subject Classification** Theory of computation → Proof theory; Computing methodologies → Concurrent programming languages

**Keywords and phrases** type-based termination, sized types, futures, concurrency, infinite proofs

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.12

**Related Version** *Extended Version*: `https://arxiv.org/abs/2105.06024`

**Funding** This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.
*Siva Somayyajula*: Partially supported by the Sansom Graduate Fellowship in Computer Science.

**Acknowledgements** We would like to thank Farzaneh Derakhshan, Klaas Pruiksma, Henry DeYoung, Ankush Das, and the anonymous reviewers for helpful discussion and suggestions regarding the contents of this paper.

## 1 Introduction

Adding (co)inductive types and terminating recursion (including productive corecursive definitions) to any programming language is a non-trivial task, since only certain recursive programs constitute valid applications of (co)induction principles. Briefly, inductive calls must occur on data smaller than the input and, dually, coinductive calls must be guarded by further codata output. In either case, we are concerned with the decrease of (co)data size – height of data and observable depth of codata – in a sequence of recursive calls. Since inferring this exactly is intractable, languages like Agda (before version 2.4) [4] and Coq [59] resort to conservative syntactic criteria like the *guardedness check*.

One solution that avoids syntactic checks is to track the flow of (co)data size at the type level with *sized types*, as pioneered by Hughes et al. [39] and further developed by others [8, 10, 2, 4]. Inductive and coinductive types are indexed by the height and observable depth of their data and codata, respectively. Consider the equirecursive type definitions in Example 1 adorned with our novel *sized type refinements*: $\mathrm{nat}[i]$ describes unary natural numbers less than or equal to $i$ and $\mathrm{stream}_A[i]$ describes infinite $A$-streams that allow the first $i+1$

© Siva Somayyajula and Frank Pfenning;
licensed under Creative Commons License CC-BY 4.0

elements to be observed before reaching potentially undefined or divergent behavior. $\oplus$ and $\&$ are respectively analogous to eager variant record and lazy record types in the functional setting.

▶ **Example 1** (Recursive types).

$$\mathrm{nat}[i] = \oplus\{\mathrm{zero} : \mathbf{1}, \mathrm{succ} : i > 0 \wedge \mathrm{nat}[i-1]\}$$
$$\mathrm{stream}_A[i] = \&\{\mathrm{head} : A, \mathrm{tail} : i > 0 \Rightarrow \mathrm{stream}_A[i-1]\}$$

Note that $\mathrm{stream}_A[i]$ is *not* polymorphic, but is parametric in the choice of $A$ for demonstrative purposes.

The phrases $\phi \wedge \ldots$ and $\phi \Rightarrow \ldots$ are *constrained types*, so that the succ branch of $\mathrm{nat}[i]$ produces a nat at height $i-1$ *when* $i > 0$ whereas the tail branch of $\mathrm{stream}_A[i]$ can produce the remainder of the stream at depth $i-1$ *assuming* $i > 0$. Starting from $\mathrm{nat}[i]$, recursing *on*, for example, $\mathrm{nat}[i-1]$ ($i > 0$ is assumed during *elimination* so that $i-1$ is well-defined) produces the size sequence $i > i-1 > i-2 > \ldots$ that eventually terminates at 0, agreeing with the (strong) induction principle for natural numbers. Dually, starting from $\mathrm{stream}_A[i]$, recursing *into* $\mathrm{stream}_A[i-1]$ (again, $i > 0$ is assumed during *introduction* so that $i-1$ is well-defined) produces the same well-founded sequence of sizes, agreeing with the coinduction principle for streams. In either case, a recursive program terminates if its call graph generates a well-founded sequence of sizes in each code path. Most importantly, the behavior of constraint conjunction and implication during elimination and introduction encodes induction and coinduction, respectively. To see how sizes are utilized in the definition of recursive programs, consider the type signatures below. We will define the code of these programs in Example 5.

▶ **Example 2** (Evens and odds I). Postponing the details of our typing judgment for the moment, the signature below describes definitions that project the even- and odd-indexed substreams (referred to by $y$) of some input stream (referred to by $x$) at half of the original depth. Note that indexing begins at zero.

$$i; \cdot; x : \mathrm{stream}_A[2i] \vdash^i y \leftarrow \mathrm{evens}\ i\ x :: (y : \mathrm{stream}_A[i])$$
$$i; \cdot; x : \mathrm{stream}_A[2i+1] \vdash^i y \leftarrow \mathrm{odds}\ i\ x :: (y : \mathrm{stream}_A[i])$$

An alternate typing scheme that hides the exact size change is shown below – given a stream of *arbitrary* depth, we may project its even- and odd-indexed substreams of arbitrary depth, too. We provide implementations for both versions in Example 5.

$$i; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash^i y \leftarrow \mathrm{evens}\ i\ x :: (y : \mathrm{stream}_A[i])$$
$$i; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash^i y \leftarrow \mathrm{odds}\ i\ x :: (y : \mathrm{stream}_A[i])$$

$\exists j.\, X[j]$ and $\forall j.\, X[j]$ denote *full* inductive and coinductive types, respectively, classifying (co)data of arbitrary size. In general, less specific type signatures are necessary when the exact size change is difficult to express at the type level [65]. For example, in relation to an input list of height $i$, the height $j$ of the output list from a list filtering function may be constrained as $j \leq i$.

Sized types are *compositional*: since termination checking is reduced to an instance of typechecking, we avoid the brittleness of syntactic termination checking. However, we find that *ad hoc* features for implementing size arithmetic in the prior work can be subsumed by

more general *arithmetic refinements* [26, 65], giving rise to our notion of sized type refinements that combine the "good parts" of modern sized type systems. First, the instances of constraint conjunction and implication to encode inductive and coinductive types, respectively, in our system are similar to the bounded quantifiers in MiniAgda [3], which gave an elegant foundation for mixed inductive-coinductive functional programming, avoiding continuity checking [2]. Unlike the prior work, however, we are able to modulate the specificity of type signatures: (slight variations of) those in Example 2 are given in $\mathsf{CIC}_{\ell}^{\widehat{}}$ [54] and MiniAgda [3, 1]. Furthermore, we avoid transfinite indices in favor of permitting some unbounded quantification (following Vezzosi [62]), achieving the effect of somewhat complicated infinite sizes without leaving finite arithmetic.

Moreover, some prior work, which is based on sequential functional languages, encodes recursion via various fixed point combinators that make both mixed inductive-coinductive programming [9] and substructural typing difficult, the latter requiring the use of the ! modality [63]. Thus, like $\mathsf{F}_{\omega}^{\mathsf{cop}}$ [4], we consider a signature of parametric recursive definitions. However, we make typing derivations for recursive programs infinitely deep by unfolding recursive calls *ad infinitum* [13, 45], which is not only more elegant than finitary typing, but also simplifies our normalization argument. To prove strong normalization, we observe that *arithmetically closed* typing derivations, which have no free arithmetic variables or constraint assumptions, can be translated to infinitely wide but finitely deep trees of a different judgment. The resulting derivations are then the induction target for our proof, leaving the option of making the original typing judgment arbitrarily rich. Thus, although our proposed language is not substructural, this result extends to programs that use their data substructurally. In short, our contributions are as follows:

1. A general system of sized types based on arithmetic refinements subsuming features of prior systems, such as the mixed inductive-coinductive types of MiniAgda [3] as well as the linear size arithmetic of $\mathsf{CIC}_{\ell}^{\widehat{}}$ [54]. Moreover, we do not depend on transfinite arithmetic.
2. The first language for mixed inductive-coinductive programming that is a subsuming paradigm [46] for futures-based functional concurrency.
3. A method for proving normalization in the presence of infinitely deep typing derivations by translation to infinitely wide but finitely deep trees. The diamond property of program reduction implies that normalization is schedule-independent, encompassing call-by-need and call-by-value strategies.

We define $\mathrm{SAX}^{\infty}$, which extends the semi-axiomatic sequent calculus (SAX) [33] with arithmetic refinements, recursion, and infinitely deep typing derivations (Section 2). Then, we define an auxiliary type system called $\mathrm{SAX}^{\omega}$ which has infinitely wide but finitely deep derivations to which we translate the derivations of $\mathrm{SAX}^{\infty}$ (Section 3). Then, we show that all $\mathrm{SAX}^{\omega}$-typed programs are strongly normalizing by a novel logical relations argument over *configurations* of processes that capture the state of a concurrent computation (Section 4).

## 2 SAX$^{\infty}$

In this section, we extend SAX [33] with recursion and arithmetic refinements in the style of Das and Pfenning [26]. SAX is a logic-based formalism and subsuming paradigm [46] for concurrent functional programming that conceives call-by-need and call-by-value strategies as particular concurrent schedules [51]. Concurrency and parallelism devices like fork/join, futures [37], and SILL-style [61] monadic concurrency can all be encoded and used side-by-side in SAX [51].

To review SAX, let us make observations about proof-theoretic *polarity*. In the sequent calculus, inference rules are either *invertible* – can be applied at any point in the proof search process, like the right rule for implication – or *noninvertible*, which can only be applied when the sequent "contains enough information," like the right rules for disjunction. Connectives that have noninvertible right rules are *positive* and those that have noninvertible left rules are *negative*. The key innovation of SAX is to replace the noninvertible rules with their axiomatic counterparts in a Hilbert-style system. Consider the following right rule for implication as well as the original left rule in the middle that is replaced with its axiomatic counterpart on the right.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to\mathrm{R} \qquad \frac{\Gamma, A \to B \vdash A \quad \Gamma, A \to B, B \vdash C}{\Gamma, A \to B \vdash C} \to\mathrm{L} \qquad \frac{}{\Gamma, A \to B, A \vdash B} \to\mathrm{L}$$

Since the axiomatic rules drop the premises of their sequent calculus counterparts, cut elimination corresponds to asynchronous communication just as the standard sequent calculus models synchronous communication [15]. In particular, SAX has a *shared memory interpretation*, mirroring the memory-based semantics of *futures* [37]. A future $x$ of type $A$ either contains an object of type $A$ or is not yet populated. A process reading from $x$ either succeeds immediately or blocks if $x$ is not yet populated. As a result, the sequent becomes the typing judgment (extended with arithmetic refinements in the style of [26]):

$$\overbrace{i, j, \ldots}^{\mathcal{V}}; \overbrace{\phi, \psi, \ldots}^{\mathcal{C}}; \overbrace{x : A, y : B, \ldots}^{\Gamma} \vdash^{\overline{e}} P :: (z : C)$$

where the *arithmetic variables* in $\mathcal{V}$ are free in the *constraints* (arithmetic formulas) in $\mathcal{C}$, the types in $\Gamma$, the *process* $P$, and type $C$; moreover, the *address variables* in $\Gamma$, which are free in $P$, stand for addresses of memory cells representing futures. In particular, $P$ reads from $x, y, \ldots$ (*sources*) and writes to $z$ (a *destination*) according to the protocols specified by $A, B, \ldots$ and $C$, respectively. $z$ is written to exactly once corresponding to the population of a future [37]. Lastly, the vector (indicated by the overline) of *arithmetic expressions* $\overline{e}$ will be used to track the sizes encountered at each recursive call as mentioned in the introduction. Now, let us examine the definitions of types and processes. For our purposes, detailed syntaxes for expressions $e$ and formulas $\phi$ are unnecessary.

▶ **Definition 3** (Type). *Types are defined by the following grammar, presupposing some mutually recursive type definitions of the form $X[\overline{i}] = A_X(\overline{i})$. Positive types (in the left column) and negative types (in the right column) are colored red and black, respectively. Recursive type names are colored blue since they take on the polarity of their definientia.*

| | | | | | |
|---|---|---|---|---|---|
| $A, B ::=$ | $\mathbf{1}$ | *unit* | \| | $X[\overline{e}]$ | *equirecursive type* |
| \| | $A \otimes B$ | *eager pair* | \| | $A \to B$ | *function* |
| \| | $\oplus\{\ell : A_\ell\}_{\ell \in S}$ | *eager variant record* | \| | $\&\{\ell : A_\ell\}_{\ell \in S}$ | *lazy record* |
| \| | $\phi \land A$ | *constraint conjunction* | \| | $\phi \Rightarrow A$ | *constraint implication* |
| \| | $\exists i.\, A(i)$ | *arithmetic dependent pair* | \| | $\forall i.\, A(i)$ | *arithmetic dep. function* |

There are eight kinds of processes: two for the structural rules (identity and cut), one for each combination of type polarity (positive or negative) and rule type (left or right), one for definition calls, and one for unreachable code.

▶ **Definition 4** (Process). *Processes are defined by the following grammar. The superscripts* R *and* W *indicating reading from or writing to a cell.*

| | | |
|---|---|---|
| $P, Q :=$ | $y^{\mathrm{W}} \leftarrow x^{\mathrm{R}}$ | *copy contents of $x$ to $y$* |
| $\mid$ | $x \leftarrow P(x); Q(x)$ | *allocate $x$, spawn $P$ to write to $x$ and concurrently proceed as $Q$, which may read from $x$* |
| $\mid$ | $x^{\mathrm{W}}.V$ | *write value $V$ to $x$* |
| $\mid$ | $\mathbf{case}\, x^{\mathrm{R}}\, K$ | *read value stored in $x$ and pass it to continuation $K$* |
| $\mid$ | $\mathbf{case}\, x^{\mathrm{W}}\, K$ | *write continuation $K$ to $x$* |
| $\mid$ | $x^{\mathrm{R}}.V$ | *read continuation stored in $x$ then pass value $V$ to it* |
| $\mid$ | $y \leftarrow f\ \overline{e}\ \overline{x}$ | *expands to $P_f(\overline{e}, \overline{x}, y)$ from a signature of mutually recursive definitions of the form $y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y)$* |
| $\mid$ | $\mathbf{impossible}$ | *unreachable code due to inconsistent arithmetic context* |

*The first two kinds of processes correspond to the identity and cut rules. Values $V$ and continuations $K$ are specified on a per-type-and-rule basis in the following two tables. Note the address variable $x$ distinguished by each rule.*

| *polarity* | *right rule* | *left rule* |
|---|---|---|
| *positive* | $x^{\mathrm{W}}.V$ | $\mathbf{case}\, x^{\mathrm{R}}\, K$ |
| *negative* | $\mathbf{case}\, x^{\mathrm{W}}\, K$ | $x^{\mathrm{R}}.V$ |

| *type(s)* | *value $V$* | *continuation $K$* |
|---|---|---|
| $\mathbf{1}$ | $\langle\rangle$ | $\langle\rangle \Rightarrow P$ |
| $\otimes, \multimap$ | $\langle y, z\rangle$ | $\langle y, z\rangle \Rightarrow P(y, z)$ |
| $\&, \oplus$ | $\ell\, y$ | $\{\ell\, y \Rightarrow P(y)\}_{\ell \in S}$ |
| $\wedge, \Rightarrow$ | $\langle *, y\rangle$ | $\langle *, y\rangle \Rightarrow P(y)$ |
| $\forall, \exists$ | $\langle e, y\rangle$ | $\langle i, y\rangle \Rightarrow P(i, y)$ |

To borrow terminology from linear logic, the "multiplicative" group ($\mathbf{1}$, $\otimes$, $\multimap$) is concerned with writing addresses, whereas the "additive" group ($\oplus$, $\&$) is concerned with writing labels and their case analysis. Constrained types read and write a placeholder $*$ indicating that a constraint is asserted or assumed. However, we will suppress instances of $*$ in the example code given, since assumptions and assertions are inferrable in the absence of consecutively alternating constraints (e.g., $\phi \wedge (\psi \Rightarrow A)$). On the other hand, the arithmetic data communicated by quantifiers are visible since inference is difficult in general [27]. Now that we are acquainted with the process syntax, let us complete Example 2.

▶ **Example 5** (Evens and odds II). Recall that we are implementing the following signature and $\mathrm{stream}_A[i] = \&\{\mathrm{head} : A, \mathrm{tail} : i > 0 \Rightarrow \mathrm{stream}_A[i-1]\}$.

$$i; \cdot; x : \mathrm{stream}_A[2i] \vdash^i y \leftarrow \mathrm{evens}\ i\ x :: (y : \mathrm{stream}_A[i])$$
$$i; \cdot; x : \mathrm{stream}_A[2i+1] \vdash^i y \leftarrow \mathrm{odds}\ i\ x :: (y : \mathrm{stream}_A[i])$$

The even-indexed substream retains the head of the input, but its tail is the odd-indexed substream of the input's tail. The odd-indexed substream, on the other hand, is simply the even-indexed substream of the input's tail. Operationally, the heads and tails of both substreams are computed on demand similar to a lazy record. Unlike their sequential counterparts, however, the recursive calls proceed concurrently due to the nature of cut. Since our examples will keep constraints implicit, we indicate when constraints are assumed or asserted inline for clarity.

$$y \leftarrow \mathrm{evens}\ i\ x = \mathbf{case}\, y^{\mathrm{W}} \{\, \mathrm{head}\ h \Rightarrow x^{\mathrm{R}}.\, \mathrm{head}\ h, \underbrace{\mathrm{tail}\, y_t}_{i>0 \text{ assumed}} \Rightarrow x_t \leftarrow \underbrace{x^{\mathrm{R}}.\, \mathrm{tail}\, x_t}_{2i>0 \text{ asserted}}\,; \underbrace{y_t \leftarrow \mathrm{odds}\,(i-1)\ x_t}_{i; i>0 \vdash i-1<i \text{ checked}} \}$$

$$y \leftarrow \mathrm{odds}\ i\ x = x_t \leftarrow \underbrace{x^{\mathrm{R}}.\, \mathrm{tail}\, x_t}_{2i+1>0 \text{ asserted}}; y \leftarrow \mathrm{evens}\ i\ x_t$$

By inlining the definition of odds in evens and vice versa, both programs terminate according to our criterion from the introduction even though odds calls evens with argument $i$. However, we sketch an alternate termination argument for similar such definitions at the end of Section 3. On the other hand, consider the alternate signature we gave.

$$i; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash^i y \leftarrow \mathrm{evens}\ i\ x :: (y : \mathrm{stream}_A[i])$$
$$i; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash^i y \leftarrow \mathrm{odds}\ i\ x :: (y : \mathrm{stream}_A[i])$$

First, we define head and tail observations on streams of arbitrary depth. Since they are not recursive, we do not bother tracking the size superscript of the typing judgment, since they can be inlined. Moreover, we take the liberty to nest values (boxed and highlighted yellow), which can be expanded into SAX [51].

$$\cdot; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash y \leftarrow \mathrm{head}\, x :: (y : A)$$
$$y \leftarrow \mathrm{head}\, x = x^{\mathrm{R}}.\boxed{\langle 0, \mathrm{head}\, y\rangle}$$
$$\cdot; \cdot; x : \forall j.\ \mathrm{stream}_A[j] \vdash y \leftarrow \mathrm{tail}\, x :: (y : \forall j.\ \mathrm{stream}_A[j])$$
$$y \leftarrow \mathrm{tail}\, x = \mathbf{case}\, y^{\mathrm{W}}\, (\langle j, y'\rangle \Rightarrow \underbrace{x^{\mathrm{R}}.\boxed{\langle j+1, \mathrm{tail}\, y'\rangle}}_{j+1>0 \text{ asserted}})$$

The implementation of odds and evens follows almost exactly as before with the above observations in place. Note that we use the abbreviation $y \leftarrow f\ \overline{e}\ \overline{x}; Q \triangleq y \leftarrow (y \leftarrow f\ \overline{e}\ \overline{x}); Q$ for convenience.

$$y \leftarrow \mathrm{evens}\ i\ x = \mathbf{case}\, y^{\mathrm{W}}\{\mathrm{head}\ h \Rightarrow y \leftarrow \mathrm{head}\, x,$$
$$\mathrm{tail}\, y_t \Rightarrow x_t \leftarrow \mathrm{tail}\, x; y_t \leftarrow \mathrm{odds}\,(i-1)\ x_t\}$$
$$y \leftarrow \mathrm{odds}\ i\ x = x_t \leftarrow \mathrm{tail}\, x; y \leftarrow \mathrm{evens}\ i\ x_t$$

Refer to Figure 1 for the full process typing judgment – we will comment on specific rules when necessary, but section 5 of [33] discusses the propositional rules more closely. In particular, the arithmetic typing rules make use of a *well-formedness judgment* $\mathcal{V}; \mathcal{C} \vdash e$ and *entailment* $\mathcal{V}; \mathcal{C} \vdash \phi$. Moreover, since the constraint rules are implicit, the noninvertible ones are not axiomatic. Most importantly, there are two rules for recursive calls; let us reproduce them below.

$$\frac{\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}} P :: (y : A)}{\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)}\ \infty \qquad \frac{\mathcal{V}; \mathcal{C} \vdash \overline{e'} < \overline{e} \quad y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \mathcal{V}; \mathcal{C}; \overline{x} : \overline{A} \vdash^{\overline{e'}}_{\infty} P_f(\overline{e'}, \overline{x}, y) :: (y : A)}{\mathcal{V}; \mathcal{C}; \Gamma, \overline{x} : \overline{A} \vdash^{\overline{e}} y \leftarrow f\ \overline{e'}\ \overline{x} :: (y : A)}\ \mathrm{call}$$

Our process typing judgment is itself *mixed inductive-coinductive* [22] – we introduce the auxiliary judgment $\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)$ that is *coinductively* generated by the $\infty$ rule (indicated by the double line). Since the premise of the call rule refers to $\mathcal{V}; \mathcal{C}; \Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)$, all valid typing derivations are trees whose infinite branches have a call-$\infty$ pair occurring infinitely often, representing the unfolding of a recursive process. At each unfolding, we check that the arithmetic arguments have decreased (from $\overline{e}$ to $\overline{e'}$) lexicographically[1] for termination.

For typechecking in finite time, restricting our type system to *circular derivations*, which can be represented as finite trees with loops, and decidable arithmetic (e.g., Presburger) is sufficient, although we do not show this formally. In short, such a restricted system can

[1] If two vectors have different lengths, then zeroes are appended to the shorter one.

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma, x : A \vdash^{\overline{e}} y^{\mathrm{W}} \leftarrow x^{\mathrm{R}} :: (y : A)}\ \mathrm{id} \qquad \frac{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} P(x) :: (x : A) \quad \mathcal{V};\mathcal{C};\Gamma, x : A \vdash^{\overline{e}} Q(x) :: (z : C)}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} x \leftarrow P(x); Q(x) :: (z : C)}\ \mathrm{cut}$$

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} x^{\mathrm{W}}.\langle\rangle :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R} \qquad \frac{\mathcal{V};\mathcal{C};\Gamma, x : \mathbf{1} \vdash^{\overline{e}} P :: (z : C)}{\mathcal{V};\mathcal{C};\Gamma, x : \mathbf{1} \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{R}}\, (\langle\rangle \Rightarrow P) :: (z : C)}\ \mathbf{1}\mathrm{L}$$

$$\frac{}{\mathcal{V};\mathcal{C};\Gamma, y : A, z : B \vdash^{\overline{e}} x^{\mathrm{W}}.\langle y, z\rangle :: (x : A \otimes B)}\ \otimes\mathrm{R} \qquad \frac{\mathcal{V};\mathcal{C};\Gamma, x : A \otimes B, y : A, z : B \vdash^{\overline{e}} P(y, z) :: (w : C)}{\mathcal{V};\mathcal{C};\Gamma, x : A \otimes B \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{R}}\, (\langle y, z\rangle \Rightarrow P(y, z)) :: (w : C)}\ \otimes\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C};\Gamma, y : A \vdash^{\overline{e}} P(y, z) :: (z : B)}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{W}}\, (\langle y, z\rangle \Rightarrow P(y, z)) :: (x : A \to B)}\ \to\mathrm{R} \qquad \frac{}{\mathcal{V};\mathcal{C};\Gamma, x : A \to B, y : A \vdash^{\overline{e}} x^{\mathrm{R}}.\langle y, z\rangle :: (z : B)}\ \to\mathrm{L}$$

$$\frac{k \in S}{\mathcal{V};\mathcal{C};\Gamma, y : A_k \vdash^{\overline{e}} x^{\mathrm{W}}.k\, y :: (x : \oplus\{\ell : A_\ell\}_{\ell \in S})}\ \oplus\mathrm{R} \qquad \frac{\{\mathcal{V};\mathcal{C};\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S}, y : A_\ell \vdash^{\overline{e}} P(y) :: (z : C)\}_{\ell \in S}}{\mathcal{V};\mathcal{C};\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S} \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{R}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (z : C)}\ \oplus\mathrm{L}$$

$$\frac{\{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} P(y) :: (y : A_\ell)\}_{\ell \in S}}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{W}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (x : \&\{\ell : A_\ell\}_{\ell \in S})}\ \&\mathrm{R} \qquad \frac{k \in S}{\mathcal{V};\mathcal{C};\Gamma, x : \&\{\ell : A_\ell\}_{\ell \in S} \vdash^{\overline{e}} x^{\mathrm{R}}.k\, y :: (y : A_k)}\ \&\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C} \vdash e}{\mathcal{V};\mathcal{C};\Gamma, y : A(e) \vdash^{\overline{e}} x^{\mathrm{W}}.\langle e, y\rangle :: (x : \exists i.\, A(i))}\ \exists\mathrm{R} \qquad \frac{\mathcal{V}, i;\mathcal{C};\Gamma, x : \exists i.\, A(i), y : A(i) \vdash^{\overline{e}} P(i, y) :: (z : C)}{\mathcal{V};\mathcal{C};\Gamma, x : \exists i.\, A(i) \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{R}}\, (\langle i, y\rangle \Rightarrow P(i, y)) :: (z : C)}\ \exists\mathrm{L}$$

$$\frac{\mathcal{V}, i;\mathcal{C};\Gamma \vdash^{\overline{e}} P(i, y) :: (y : A(i))}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{W}}\, (\langle i, y\rangle \Rightarrow P(i, y)) :: (x : \forall i.\, A(i))}\ \forall\mathrm{R} \qquad \frac{\mathcal{V};\mathcal{C} \vdash e}{\mathcal{V};\mathcal{C};\Gamma, x : \forall i.\, A(i) \vdash^{\overline{e}} x^{\mathrm{R}}.\langle e, y\rangle :: (y : A(e))}\ \forall\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C} \vdash \phi}{\mathcal{V};\mathcal{C};\Gamma, y : A \vdash^{\overline{e}} x^{\mathrm{W}}.\langle *, y\rangle :: (x : \phi \wedge A)}\ \wedge\mathrm{R} \qquad \frac{\mathcal{V};\mathcal{C}, \phi;\Gamma, x : \phi \wedge A, y : A \vdash^{\overline{e}} P(y) :: (z : C)}{\mathcal{V};\mathcal{C};\Gamma, x : \phi \wedge A \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (z : C)}\ \wedge\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C}, \phi;\Gamma \vdash^{\overline{e}} P(y) :: (y : A)}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} \mathbf{case}\, x^{\mathrm{W}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (x : \phi \Rightarrow A)}\ \Rightarrow\mathrm{R} \qquad \frac{\mathcal{V};\mathcal{C} \vdash \phi}{\mathcal{V};\mathcal{C};\Gamma, x : \phi \Rightarrow A \vdash^{\overline{e}} x^{\mathrm{R}}.\langle *, y\rangle :: (y : A)}\ \Rightarrow\mathrm{L}$$

$$\frac{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}} P :: (y : A)}{\mathcal{V};\mathcal{C};\Gamma \vdash^{\overline{e}}_{\infty} P :: (y : A)}\ \infty \qquad \frac{\mathcal{V};\mathcal{C} \vdash \overline{e'} < \overline{e} \quad y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \mathcal{V};\mathcal{C};\overline{x} : \overline{A} \vdash^{\overline{e'}}_{\infty} P_f(\overline{e'}, \overline{x}, y) :: (y : A)}{\mathcal{V};\mathcal{C};\Gamma, \overline{x} : \overline{A} \vdash^{\overline{e}} y \leftarrow f\ \overline{e'}\ \overline{x} :: (y : A)}\ \mathrm{call}$$

**Figure 1** $\mathrm{SAX}^\infty$ Typing.

be put in correspondence with a finitary system that detects said loops [13, 21, 51] and arithmetic constraint obligations can be discharged mechanically [25]. In Example 22 in the appendix, we show a hypothetical instance of typechecking (note that we use "$D \in J$" to indicate a derivation $D$ of the judgment $J$). Now, consider the following example that demonstrates a use case of mixed induction-coinduction in concurrency.

▶ **Example 6** **(Left-fair streams).** Let us define the mixed inductive-coinductive type $\mathrm{lfair}_{A,B}[i, j]$ of *left-fair streams* [9]: infinite $A$-streams where each element is separated by finitely many elements in $B$. Once again, these types are *not* polymorphic, but are parametric in the choice of $A$ and $B$ for demonstration.

$$\mathrm{lfair}_{A,B}[i, j] = \oplus\{\mathrm{now} : \&\{\mathrm{head} : A, \mathrm{tail} : \mathrm{lfair}^{\nu}_{A,B}[i, j]\}, \mathrm{later} : B \otimes \mathrm{lfair}^{\mu}_{A,B}[i, j]\}$$
$$\mathrm{lfair}^{\nu}_{A,B}[i, j] = i > 0 \Rightarrow \exists j'.\ \mathrm{lfair}_{A,B}[i - 1, j']$$
$$\mathrm{lfair}^{\mu}_{A,B}[i, j] = j > 0 \wedge \mathrm{lfair}_{A,B}[i, j - 1]$$

In particular, $i$ bounds the observation depth of the $A$-stream whereas $j$ bounds the height of the $B$-list in between consecutive $A$ elements. Thus, this type is defined by lexicographic induction on $(i, j)$. First, the provider may offer an element of $A$, in which case the observation depth of the stream decreases from $i$ to $i - 1$ (in the coinductive part, $\mathrm{lfair}^{\nu}_{A,B}[i, j]$). As a

result, $j$ may be "reset" as an arbitrary $j'$. On the other hand, if an element of "padding" in $B$ is offered, then the depth $i$ does not change. Rather, the height of the $B$-list decreases from $j$ to $j-1$ (in the inductive part, $\mathrm{lfair}^{\mu}_{A,B}[i,j]$). By using left-fair streams, we can model processes that permit some timeout behavior but are eventually productive, since consecutive elements of type $A$ are interspersed with only finitely many timeout acknowledgements of type $B$. Armed with this type, we can define a *projection* operation [9] that removes all of a left-fair stream's timeout acknowledgements concurrently, returning an $A$-stream. For brevity, we nest patterns (boxed and highlighted yellow), which can be expanded into nested matches [51].

$$i, j; \cdot; x : \mathrm{lfair}_{A,B}[i,j] \vdash^{(i,j)} y \leftarrow \mathrm{proj}\,(i,j)\ x :: (y : \mathrm{stream}_A[i])$$

$$y \leftarrow \mathrm{proj}\,(i,j)\ x =$$

$$\mathbf{case}\, x^{\mathrm{R}}\, (\mathrm{now}\, s \Rightarrow \mathbf{case}\, y^{\mathrm{W}} (\mathrm{head}\, h \Rightarrow s^{\mathrm{R}}.\,\mathrm{head}\, h,$$

$$\underbrace{\mathrm{tail}\, t}_{i>0 \text{ assumed}} \Rightarrow u \leftarrow s^{\mathrm{R}}.\,\mathrm{tail}\, u;$$

$$\mathbf{case}\, u^{\mathrm{R}}\, (\underbrace{\langle j', x' \rangle}_{i>0 \text{ asserted}} \Rightarrow \underbrace{t \leftarrow \mathrm{proj}\,(i-1, j')\ x'}_{i,j,j';i>0 \vdash (i-1,j')<(i,j) \text{ checked}} )),$$

$$\underbrace{\boxed{\mathrm{later}\langle b, x' \rangle}}_{j>0 \text{ assumed}} \Rightarrow \underbrace{y \leftarrow \mathrm{proj}\,(i, j-1)\ x'}_{i,j;j>0 \vdash (i,j-1)<(i,j) \text{ checked}} )$$

## 3 SAX$^{\omega}$

Even without presenting the operational semantics yet, it is unclear how to prove normalization of program reduction in the presence of infinite typing derivations. As a result, we give a purely inductive process typing called SAX$^{\omega}$ with the judgment $\Gamma \vdash^{\omega} P :: (x : A)$ (selected rules in Figure 2 with the rest in Figure 5). By dropping the arithmetic and constraint contexts, the rules $\exists\mathrm{L}^{\omega}$ and $\forall\mathrm{R}^{\omega}$ have one premise per natural number $n$ instead of introducing a new arithmetic variable (like the $\omega$-rule of arithmetic [40]). Moreover, the premises of $\wedge\mathrm{L}^{\omega}$ and $\Rightarrow\mathrm{R}^{\omega}$ assume the closed constraint $\phi$ (which has no free arithmetic variables) holds at the meta level instead of adding it to a constraint context.

Most importantly, the call rule does not refer to a coinductively-defined auxiliary judgment, because in the absence of free arithmetic variables, the tracked size arguments decrease from some $\overline{n}$ to $\overline{n'}$ to etc. Since the lexicographic order on fixed-length natural number vectors is well-founded, this sequence necessarily terminates. To rephrase: the exact number of recursive calls is known. While this system is impractical for type checking, we can translate arithmetically closed SAX$^{\infty}$ derivations to SAX$^{\omega}$ derivations. In fact, any SAX$^{\infty}$ derivation can be made arithmetically closed by substituting each of its free arithmetic variables for numbers that validate (and therefore discharge) its constraints. By trading infinitely deep derivations for infinitely wide but finitely deep ones, we may complete a logical relations argument by induction over a SAX$^{\omega}$ derivation. Thus, let us examine the translation theorem.

▶ **Theorem 7** (Translation). *If $D \in \cdot; \cdot; \Gamma \vdash^{\overline{n}} P :: (x : A)$, then $\Gamma \vdash^{\omega} P :: (x : A)$.*

**Proof.** By lexicographic induction on $(\overline{n}, D)$, we cover the important cases.

1. When $D$ ends in $\exists\mathrm{L}$ or $\forall\mathrm{R}$, its subderivation $D'$ introduces a fresh arithmetic variable $i$. The $m^{\mathrm{th}}$ premise of the corresponding SAX$^{\omega}$ rules $\exists\mathrm{L}^{\omega}$ and $\forall\mathrm{R}^{\omega}$ are fulfilled by induction on $(\overline{n}, [m/i]D')$.

$$\frac{}{\Gamma, y : A, z : B \vdash^\omega x^{\mathrm{W}}.\langle y, z\rangle :: (x : A \otimes B)} \otimes\mathrm{R}^\omega \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash^\omega P(y,z) :: (w : C)}{\Gamma, x : A \otimes B \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle y, z\rangle \Rightarrow P(y,z)) :: (w : C)} \otimes\mathrm{L}^\omega$$

$$\frac{}{\Gamma, y : A(n) \vdash^\omega x^{\mathrm{W}}.\langle n, y\rangle :: (x : \exists i.\, A(n))} \exists\mathrm{R}^\omega \qquad \frac{\Gamma, x : \exists i.\, A(i), y : A(n) \vdash^\omega P(n,y) :: (z : C) \text{ for all } n \in \mathbb{N}}{\Gamma, x : \exists i.\, A(i) \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle i, y\rangle \Rightarrow P(i,y)) :: (z : C)} \exists\mathrm{L}^\omega$$

$$\frac{\Gamma \vdash^\omega P(n,y) :: (y : A(n)) \text{ for all } n \in \mathbb{N}}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle i, y\rangle \Rightarrow P(i,y)) :: (x : \forall i.\, A(i))} \forall\mathrm{R}^\omega \qquad \frac{}{\Gamma, x : \forall i.\, A(i) \vdash^\omega x^{\mathrm{R}}.\langle n, y\rangle :: (y : A(n))} \forall\mathrm{L}^\omega$$

$$\frac{\cdot ; \cdot \vdash \phi}{\Gamma, y : A \vdash^\omega x^{\mathrm{W}}.\langle *, y\rangle :: (x : \phi \wedge A)} \wedge\mathrm{R}^\omega \qquad \frac{\Gamma, x : \phi \wedge A, y : A \vdash^\omega P(y) :: (z : C) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \wedge A \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (z : C)} \wedge\mathrm{L}^\omega$$

$$\frac{\Gamma \vdash^\omega P(y) :: (y : A) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (x : \phi \Rightarrow A)} \Rightarrow\mathrm{R}^\omega \qquad \frac{\cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \Rightarrow A \vdash^\omega x^{\mathrm{R}}.\langle *, y\rangle :: (y : A)} \Rightarrow\mathrm{L}^\omega$$

$$\text{(no rule for } \mathbf{impossible}) \qquad \frac{y \leftarrow f\ \bar{i}\ \overline{x} = P_f(\bar{i}, \overline{x}, y) \quad \overline{x} : \overline{A} \vdash^\omega P_f(\overline{n}, \overline{x}, y) :: (y : A)}{\Gamma, \overline{x} : \overline{A} \vdash^\omega y \leftarrow f\ \overline{n}\ \overline{x} :: (y : A)} \mathrm{call}^\omega$$

**Figure 2** Selected SAX$^\omega$ Typing Rules.

2. Analogously, when $D$ ends in $\wedge$L or $\Rightarrow$R, its subderivation $D'$ assumes $\phi$. The premises of the corresponding SAX$^\omega$ rules $\wedge$L$^\omega$ and $\Rightarrow$R$^\omega$ assume $E \in \cdot ; \cdot \vdash \phi$, so we finish by induction on $(\overline{n}, E \cdot D')$ where $E \cdot D'$ cuts $\phi$ out of $D'$.
3. Finally, assume $D$ ends in the call rule with subderivation $D'$. By inversion, $D'$ ends in the $\infty$ rule with subderivation $D''$. Although $D''$ may be larger than $D$, we have some new arithmetic arguments $\overline{n'} < \overline{n}$. Thus, we are done by induction on $(\overline{n'}, D'')$ then the SAX$^\omega$ call rule. ◀

As we mentioned in the introduction, we can make the SAX$^\infty$ judgment arbitrarily rich to support more complex patterns of recursion. As long as derivations in that system can be translated to SAX$^\omega$, the logical relations argument over SAX$^\omega$ typing that we detail in Section 4 does not change. For example, consider the following additions.

1. *Multiple blocks*: To support multiple blocks of definitions, we may simply impose the requirement that mutual recursion may not occur *across* blocks. In other words, the call graph *across* blocks is directed acyclic, imposing a well-founded order on definition names: $g < f$ iff $f$ calls $g$. As a result, translation of the definition $f$ may proceed by lexicographic induction on $(f, \overline{n}, D)$. For example, let $f$ call $g$. If $g$ is defined in a different block than $f$, then the arithmetic arguments it applies $(\overline{n})$ may increase. Otherwise, $\overline{n}$ must decrease, since $g$ is "equal" to $f$ (in this order).
2. *Mutual recursion with priorities*: Definitions in a block can be ordered by *priority*: if $g < f$, then $f$ can call $g$ with arguments of the same size. In Example 5, odds calls evens with arguments of the same size but evens calls odds with arguments of lesser size. As a result, evens < odds. If $<$ is well-founded (like in this example), then translation of $f$ may proceed by lexicographic induction on $(\overline{n}, f, D)$.

# 4 Semantics and Normalization

In this section, we will give an operational semantics for *configurations* of processes. Then, we will show that all SAX$^\omega$-typed processes are strongly normalizing. Program execution based on processes alone is impractical, because cut elimination only facilitates communication

between two processes at a time. Thus, DeYoung et al. [33] define programs in SAX as *configurations* of simultaneously executing processes and the memory cells with which they communicate. Relatedly, the metatheory of the $\pi$-calculus must be defined up-to structural congruence to achieve a similar effect [53].

▶ **Definition 8** (Configuration). *Let* $a, b, c, \ldots \in \mathsf{Addr}$ *be* cell addresses *and* $W := V \mid K$*. A* configuration $C$ *is defined by the following grammar.*

$$\begin{array}{ll} C := \cdot & \textit{empty configuration} \\ \quad \mid \mathrm{proc}\, a\, P & \textit{process } P \textit{ writing to cell addressed by } a \\ \quad \mid !\mathrm{cell}\, a\, W & \textit{persistent (marked with !) cell addressed by } a \textit{ with contents } W \\ \quad \mid C, C & \textit{join of two configurations} \end{array}$$

*$C$ denotes a multiset of* objects *(processes and cells), so the join and empty rules form a commutative monoid. However, we also require that an address refers to at most one object in $C$. Lastly, a configuration $F$ is* final *iff it only contains (persistent) cells.*

Now, let $\Gamma$ and $\Delta$ be contexts that associate cell addresses to types. The configuration typing judgment given in Figure 3, $\Gamma \vdash C :: \Delta$, means that the objects in $C$ are well-typed with sources in $\Gamma$ and destinations in $\Delta$ (note that we are allowing the process typing judgment to use addresses in place of address variables). Notice that the typing rules preserve the invariant $\Gamma \subseteq \Delta$ thanks to the persistence of memory cells.

$$\frac{\Gamma \vdash^{\omega} P :: (a : A)}{\Gamma \vdash \mathrm{proc}\, a\, P :: (\Gamma, a : A)}\ \mathrm{proc} \qquad \frac{\Gamma \vdash^{\omega} a^{\mathrm{W}}.V :: (a : A)}{\Gamma \vdash !\mathrm{cell}\, a\, V :: (\Gamma, a : A)}\ !\mathrm{cell}_V$$

$$\frac{\Gamma \vdash^{\omega} \mathbf{case}\, a^{\mathrm{W}}\, K :: (a : A)}{\Gamma \vdash !\mathrm{cell}\, a\, K :: (\Gamma, a : A)}\ !\mathrm{cell}_K \qquad \frac{}{\Gamma \vdash \cdot :: \Gamma}\ \mathrm{empty} \qquad \frac{\Gamma \vdash C :: \Gamma' \quad \Gamma' \vdash C' :: \Delta}{\Gamma \vdash C, C' :: \Delta}\ \mathrm{join}$$

**Figure 3** Configuration Typing.

Configuration reduction $\to$ is given as *multiset rewriting rules* [17] in Figure 4, which replace any subset of a configuration matching the left-hand side with the right-hand side. However, ! indicates objects that persist across reductions. Principal cuts encountered in a configuration are resolved by passing a value to a continuation also given in Figure 4 as the relation $V \triangleright K = P$.

$$\begin{array}{rl} !\mathrm{cell}\, a\, W, \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) & \to\ !\mathrm{cell}\, b\, W \\ \mathrm{proc}\, c\, (x \leftarrow P(x); Q(x)) & \to \\ & \mathrm{proc}\, a\, (P(a)), \mathrm{proc}\, c\, (Q(a)) \text{ where } a \text{ is fresh} \\ !\mathrm{cell}\, a\, K, \mathrm{proc}\, c\, (a^{\mathrm{R}}.V) & \to \mathrm{proc}\, c\, (V \triangleright K) \\ !\mathrm{cell}\, a\, V, \mathrm{proc}\, c\, (\mathbf{case}\, a^{\mathrm{R}}\, K) & \to \mathrm{proc}\, c\, (V \triangleright K) \\ \mathrm{proc}\, a\, (a \leftarrow f\ \overline{n}\ \overline{b}) & \to \mathrm{proc}\, a\, (P_f(\overline{n}, \overline{b}, a)) \\ \mathrm{proc}\, a\, (a^{\mathrm{W}}.V) & \to\ !\mathrm{cell}\, a\, V \\ \mathrm{proc}\, a\, (\mathbf{case}\, a^{\mathrm{W}}\, K) & \to\ !\mathrm{cell}\, a\, K \end{array}$$

$$\begin{array}{rl} \langle\rangle \triangleright \langle\rangle \Rightarrow P &= P \\ \langle a, b\rangle \triangleright (\langle x, y\rangle \Rightarrow P(x, y)) &= P(a, b) \\ k\, a \triangleright \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S} &= P_k(a) \\ \langle n, a\rangle \triangleright (\langle i, x\rangle \Rightarrow P(i, x)) &= P(n, a) \\ \langle *, a\rangle \triangleright (\langle *, x\rangle \Rightarrow P(x)) &= P(a) \end{array}$$

**Figure 4** Operational Semantics.

The first rule for $\to$ corresponds to the identity rule and copies the contents of one cell into another. The second rule, which is for cut, models computing with futures [37]: it allocates a new cell to be populated by the newly spawned $P$. Concurrently, $Q$ may read from said new cell, which blocks if it is not yet populated. The third and fourth rules resolve principal cuts by passing a value to a continuation, whereas the fifth one resolves definition calls. Lastly, the final two rules perform the action of writing to a cell.

Now, we are ready to prove normalization. Relatedly, refer to Das and Pfenning [25] for a proof of type safety for a session type system with arithmetic refinements. In contrast to the normalization proof for base SAX [33], we explicitly construct a model of SAX in sets of terminating configurations, also known as *semantic typing* [5, 38]. This leaves open several possibilities – for example, we could reason about programs that fail to syntactically typecheck [41, 34] or analyze fixed points of semantic type constructors. Our approach mirrors that for natural deduction:

1. We define semantic types: sets of terminating configurations with the necessary properties to prove normalization (see reducibility candidates [36]).
2. We show that semantic versions of the syntactic typing rules of processes, objects, and configurations are admissible in this model.
3. This culminates in a fundamental theorem of the logical relation that translates syntactic types to semantic ones. Weak normalization for *closed* configurations (where $\cdot \vdash C :: \Delta$) is a corollary.
4. Strong normalization of arbitrary configurations (where $\Gamma \vdash C :: \Delta$) is a corollary of the fundamental theorem as well as a weak form of the diamond property [6].

Now, let us begin with the definition of semantic type.

▶ **Definition 9** (Semantic type). *A* semantic type $\mathscr{A}, \mathscr{B}, \ldots \in \textbf{Sem}$ *is a set of pairs of addresses and final configurations, writing* $F \in [a : \mathscr{A}]$ *for* $(a; F) \in \mathscr{A}$*, such that if* $F \in [a : \mathscr{A}]$*, then:*

1. Inversion*:* $!\text{cell}\, a\, W \in F$ *for some* $W$*.*
2. Contraction*:* $!\text{cell}\, b\, W \in [b : \mathscr{A}]$ *for all* $b \in \mathsf{Addr}$ *(*$W$ *is from above).*
3. Weakening*:* $F, F' \in [a : \mathscr{A}]$ *for all* $F'$*.*

*Let* $\to^*$ *be multi-step reduction and* $C \in [\![a : \mathscr{A}]\!]$ *iff* $C \to^* F$ *and* $F \in [a : \mathscr{A}]$*.*

Conditions 1 and 2 are required to reproduce the identity rule semantically, but condition 3 is a symptom of working in a concurrent setting: we need to aggregate the semantic type ascriptions of different sub-configurations. In the next definition, we quickly define each semantic type in **boldface** based on its syntactic counterpart.

▶ **Definition 10** (Semantic types).

1. $F \in [a : \mathbb{1}] \triangleq F = F', !\text{cell}\, a\, \langle\rangle$.
2. $F \in [c : \mathscr{A} \otimes \mathscr{B}] \triangleq F = F', !\text{cell}\, c\, \langle a, b\rangle$ *where* $F' \in [a : \mathscr{A}]$ *and* $F' \in [b : \mathscr{B}]$*.*
3. $F \in [c : \mathscr{A} \multimap \mathscr{B}] \triangleq !\text{cell}\, c\, K \in F$ *for some* $K$ *and* $F, F', \text{proc}\, b\, (c^{\mathrm{R}}.\langle a, b\rangle) \in [\![b : \mathscr{B}]\!]$ *for all* $a, b, F'$ *such that* $F, F' \in [a : \mathscr{A}]$*.*
4. $F \in [b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq F = F', !\text{cell}\, b\, (k\, a)$ *and* $F' \in [a : \mathscr{A}_k]$ *for some* $k \in S$*.*
5. $F \in [b : \&\{\ell : \mathscr{A}_\ell\}_{\ell \in S}] \triangleq !\text{cell}\, b\, K \in F$ *for some* $K$ *and* $F, \text{proc}\, a\, (b^{\mathrm{R}}.k\, a) \in [\![a : \mathscr{A}_k]\!]$ *for all* $k \in S, a \in \mathsf{Addr}$*.*

*Assume* $\mathscr{F} : \mathbb{N} \to \textbf{Sem}$ *and* $\phi$ *is a closed constraint.*

1. $F \in [b : \exists\mathscr{F}] \triangleq F = F', !\text{cell}\, b\, \langle n, a\rangle$ *and* $F' \in [a : \mathscr{F}(n)]$*.*
2. $F \in [b : \forall\mathscr{F}] \triangleq F, \text{proc}\, a\, (b^{\mathrm{R}}.\langle n, a\rangle) \in [\![a : \mathscr{F}(n)]\!]$ *for all* $a \in \mathsf{Addr}, n \in \mathbb{N}$*.*
3. $F \in [b : \phi \wedge \mathscr{A}] \triangleq$ *where* $F = F', !\text{cell}\, b\, \langle *, a\rangle$*,* $\cdot; \cdot \vdash \phi$*, and* $F' \in [a : \mathscr{A}]$*.*
4. $F \in [b : \phi \Rightarrow \mathscr{A}] \triangleq$ *if* $\cdot; \cdot \vdash \phi$*, then* $F, \text{proc}\, a\, (b^{\mathrm{R}}.\langle *, a\rangle) \in [\![a : \mathscr{A}]\!]$ *for all* $a \in \mathsf{Addr}$*.*

Positive semantic types are defined by *intension* – the contents of a particular cell – whereas negative semantic types are defined by *extension* – how interacting with a configuration produces the desired result. Analogously for the $\lambda$-calculus, the semantic positive product is defined as containing pairs of normalizing terms, whereas the semantic function space contains all terms that normalize under application [36, 4]. Now, to state the semantic typing rules, we need to define the *semantic typing judgment.*

▶ **Definition 11** (Semantic typing judgment). *Let $\mathbf{\Gamma}$ and $\mathbf{\Delta}$ be contexts associating cell addresses to semantic types.*

1. $F \in [\mathbf{\Gamma}] \triangleq F \in [a : \mathscr{A}]$ *for all* $a : \mathscr{A} \in \mathbf{\Gamma}$.
2. $C \in [\![\mathbf{\Gamma}]\!] \triangleq C \mapsto^* F$ *and* $F \in [\mathbf{\Gamma}]$.
3. $\mathbf{\Gamma} \vDash C :: \mathbf{\Delta} \triangleq$ *for all* $F \in [\mathbf{\Gamma}]$*, we have* $F, C \in [\![\mathbf{\Delta}]\!]$.

In natural deduction, the equivalent judgment $\Gamma \vDash e : \mathscr{A}$ is defined by quantifying over all closing value substitutions $\sigma$ with domain $\Gamma$, then stating $\sigma(e) \in \mathscr{A}$. Similarly, we ask whether the configuration $C$ terminates at the desired semantic type(s) when "closed" by a final configuration $F$ providing all the sources from which $C$ reads. Immediately, we reproduce the standard backwards closure result.

▶ **Lemma 12** (Backward closure). *If $C \to^* C'$ and $\mathbf{\Gamma} \vDash C' :: \mathbf{\Delta}$, then $\mathbf{\Gamma} \vDash C :: \mathbf{\Delta}$.*

We are finally ready to prove a representative sample of semantic typing rules, all of which are in Figure 6 (the dashed lines indicate that they are admissible rules). Afterwards, we can tackle objects and configurations. As we promised, conditions 1 and 2 are used for the admissibility of the identity rule.

▶ **Lemma 13** (id). $\mathbf{\Gamma}, a : \mathscr{A} \vDash \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) :: (b : \mathscr{A})$

**Proof.** Assuming $F \in [\mathbf{\Gamma}, a : \mathscr{A}]$, we want to show $F, \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) \in [\![b : \mathscr{A}]\!]$. By condition 1, $!\mathrm{cell}\, a\, W \in F$. By condition 2, $F, !\mathrm{cell}\, b\, W \in [b : \mathscr{A}]$. Since $F, \mathrm{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) \to F, !\mathrm{cell}\, b\, W$, we are done by Lemma 12. ◀

The reader may have noticed that each semantic type's definition encodes its own noninvertible rule, which makes the admissibility of rules like $\otimes$R immediate. Invertible rules require more effort; consider $\otimes$L below.

▶ **Lemma 14** ($\otimes$L). *If $\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \mathrm{proc}\, d\, (P(a, b)) :: (d : \mathscr{C})$, then $\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B} \vdash \mathrm{proc}\, d\, (\mathbf{case}\, c^{\mathrm{R}}\, (\langle x, y\rangle \Rightarrow P(x, y))) :: (d : \mathscr{C})$.*

**Proof.** Assuming $F \in [\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}]$, we want to show that $F, \mathrm{proc}\, d\, (\mathbf{case}\, c^{\mathrm{R}}\, (\langle x, y\rangle \Rightarrow P(x, y))) \in [\![d : \mathscr{C}]\!]$. Since $F \in [c : \mathscr{A} \otimes \mathscr{B}]$, we have $F = F', !\mathrm{cell}\, c\, \langle a, b\rangle$ where $F' \in [a : \mathscr{A}]$ and $F' \in [b : \mathscr{B}]$. As a result, both $F \in [a : \mathscr{A}]$ and $F \in [b : \mathscr{B}]$ by condition 3. In sum, $F \in [\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B}]$, so by the premise, $F, \mathrm{proc}\, d\, (P(a, b)) \in [\![d : \mathscr{C}]\!]$. Since $F, \mathrm{proc}\, d\, (\mathbf{case}\, c^{\mathrm{R}}\, (\langle x, y\rangle \Rightarrow P(x, y))) \to F, \mathrm{proc}\, d\, (P(a, b))$, we are done by Lemma 12. ◀

Ironically, the persistence of a cell from the conclusion to the premise of a rule, which encodes contraction, is justified via condition 3 (semantic weakening). On the other hand, the identity rule, which "bakes in" weakening, is justified via condition 2 (semantic contraction). Now, to prove semantic object typing rules, we need a *logical relation* that interprets syntactic types as semantic ones.

▶ **Definition 15** (Logical relation). *We define a logical relation* $(\!|A|\!)_n$ *by lexicographic induction on* $(n, A)$ *that sends arithmetically closed types to semantic ones. At a recursive type, n is stepped down to allow A to potentially grow larger. Note that* $\lambda$ *marks a meta-level anonymous function and that* $\phi$ *is closed.*

$$
\begin{array}{lll}
(\!|\mathbf{1}|\!)_n \triangleq \mathbb{1} & (\!|A \otimes B|\!)_n \triangleq (\!|A|\!)_n \otimes (\!|B|\!)_n & (\!|A \to B|\!)_n \triangleq (\!|A|\!)_n \to (\!|B|\!)_n \\
 & (\!|\oplus\{\ell : A_\ell\}_{\ell \in S}|\!)_n \triangleq \oplus\{\ell : (\!|A_\ell|\!)_n\}_{\ell \in S} & (\!|\&\{\ell : A_\ell\}_{\ell \in S}|\!)_n \triangleq \&\{\ell : (\!|A_\ell|\!)_n\}_{\ell \in S} \\
(\!|X[\overline{m}]|\!)_0 \triangleq \emptyset & (\!|\forall i.\, A(i)|\!)_n \triangleq \forall(\lambda m.\, (\!|A(m)|\!)_n) & (\!|\exists i.\, A(i)|\!)_n \triangleq \exists(\lambda m.\, (\!|A(m)|\!)_n) \\
(\!|X[\overline{m}]|\!)_{n+1} \triangleq (\!|A_X[\overline{m}]|\!)_n & (\!|\phi \wedge A|\!)_n \triangleq \phi \wedge (\!|A|\!)_n & (\!|\phi \Rightarrow A|\!)_n \triangleq \phi \Rightarrow (\!|A|\!)_n
\end{array}
$$

*The index n is merely a technical device for defining the logical relation – it is* not *a step index. Now, let* $F \in (\!|A|\!) \triangleq F \in (\!|A|\!)_n$ *for some n.* $(\!|\cdot|\!)$ *is then extended to contexts* $\Gamma$ *and* $\Delta$ *in the obvious way.*

▶ **Lemma 16** (Semantic object typing).
1. *If* $D \in \Gamma \vdash^\omega a^{\mathrm{W}}.V :: (a : A)$*, then* $(\!|\Gamma|\!) \vDash \,!\mathrm{cell}\, a\, V :: (a : (\!|A|\!))$.
2. *If* $D \in \Gamma \vdash^\omega \mathbf{case}\, a^{\mathrm{W}}\, K :: (a : A)$*, then* $(\!|\Gamma|\!) \vDash \,!\mathrm{cell}\, a\, K :: (a : (\!|A|\!))$.
3. *If* $D \in \Gamma \vdash^\omega P :: (a : A)$*, then* $(\!|\Gamma|\!) \vDash \mathrm{proc}\, a\, P :: (a : (\!|A|\!))$.

**Proof.** Part 1 follows by case analysis on $D$ applying the relevant semantic typing rules, like $\otimes$R for $\otimes\mathrm{R}^\omega$. We prove parts 2 and 3 simultaneously by lexicographic induction on $D$ then the part number. That is, part 2 refers to part 3 on the typing subderivation for the process contained in $K$ (like $\to\mathrm{R}^\omega$). In part 3, if $P$ reads a cell (like $\to\mathrm{L}^\omega$ or $\otimes\mathrm{L}^\omega$), then we invoke the relevant semantic typing rule. If $P$ writes a continuation $K$, then $\mathrm{proc}\, a\, (\mathbf{case}\, a^W\, K) \to \,!\mathrm{cell}\, a\, K$, so we invoke part 2 on $D$ and conclude by Lemma 12. Writing a value follows symmetrically, invoking part 1. ◀

The reader may have already noticed that there is a disconnect: the conclusions of our semantic object typing rules have a single succedent, e.g., $(\!|\Gamma|\!) \vDash \mathrm{proc}\, a\, P :: (a : (\!|A|\!))$, but its syntactic counterpart factors sources through: $\Gamma \vdash \mathrm{proc}\, a\, P :: (\Gamma, a : A)$. The following lemma recovers this information as a consequence of memory cell persistence.

▶ **Lemma 17** (Recall). *If* $\mathbf{\Gamma} \vDash C :: \mathbf{\Delta}$*, then* $\mathbf{\Gamma} \vDash C :: \mathbf{\Gamma}, \mathbf{\Delta}$.

Now that processes and objects have been resolved, it remains to derive the semantic configuration typing rules.

▶ **Lemma 18** (Semantic configuration typing).
1. *Empty:* $\mathbf{\Gamma} \vDash \cdot :: \mathbf{\Gamma}$
2. *Join: If* $\mathbf{\Gamma} \vDash C :: \mathbf{\Gamma}'$ *and* $\mathbf{\Gamma}' \vDash C' :: \mathbf{\Delta}$*, then* $\mathbf{\Gamma} \vDash C, C' :: \mathbf{\Delta}$.

The previous lemmas establish the fundamental theorem of the logical relation, of which weak normalization of closed configurations is a corollary.

▶ **Theorem 19** (Fundamental theorem). *If* $D \in \Gamma \vdash C :: \Delta$*, then* $(\!|\Gamma|\!) \vDash C :: (\!|\Delta|\!)$.

**Proof.** By induction on $D$, the empty and join cases are discharged by Lemma 18. The object typing cases are covered by Lemma 16 then Lemma 17. ◀

Strong normalization follows from weak normalization as well as a weak form of the diamond property (as follows) [6]. As we mentioned in the introduction, the latter implies that normalization is independent of how processes are scheduled.

▶ **Theorem 20** (Diamond property). *Let $C_1 \sim C_2$ iff $C_1$ is equal to $C_2$ up-to renaming of addresses. Assume $\Gamma \vdash C :: \Delta$, $C \to C_1$, and $C \to C_2$ where $C_1 \not\sim C_2$. Then, $C_1 \to C_1'$ and $C_2 \to C_2'$ such that $C_1' \sim C_2'$.*

**Proof.** The proof follows that of theorem 10 (the diamond property) in [33], as the only relevant addition is the unfolding of recursive definitions. ◀

▶ **Theorem 21** (Strong normalization). *If $\Gamma \vdash C :: \Delta$, then there are no infinite reduction sequences beginning with $C$.*

## 5 Related Work

Our system is closely related to the sequential functional language of Lepigre and Raffalli [45], which utilizes circular typing derivations for a sized type system with mixed inductive-coinductive types, also avoiding continuity checking. In particular, their well-foundedness criterion on circular proofs seems to correspond to our checking that sizes decrease between recursive calls. However, they encode recursion using a fixed point combinator and use transfinite size arithmetic, both of which we avoid as we explained in the introduction. Moreover, our metatheory, which handles *infinite* typing derivations (via mixed induction-coinduction at the meta level), seems to be both simpler and more general since it does not have to explicitly rule out non-circular derivations. Nevertheless, we are interested in how their innovations in polymorphism and Curry-style subtyping can be integrated into our system, especially the ability to handle programs not annotated with sizes.

**Sized types.** Sized types are a type-oriented formulation of size-change termination [44] for rewrite systems [60, 12]. Sized (co)inductive types [8, 10, 2, 4] gave way to sized mixed inductive-coinductive types [3, 4]. In parallel, linear size arithmetic for sized inductive types [19, 64, 11] was generalized to support coinductive types as well [54]. We present, to our knowledge, the first sized type system for a concurrent programming language as well as the first system to combine both features from above. As we mentioned in the introduction, we use unbounded quantification [62] in lieu of transfinite sizes to represent (co)data of arbitrary height and depth. However, the state of the art [3, 4, 18] supports polymorphic, higher-kinded, and dependent types, which we aim to incorporate in future work.

**Size inference.** Our system keeps constraints implicit but arithmetic data explicit at the process level in agreement with observations made about constraint and arithmetic term reconstruction in a session-typed calculus [27]. On the other hand, systems like $\mathsf{CIC}\widehat{_\ell}$ [54] and $\mathsf{CIC}\widehat{_*}$ [18] have comprehensive *size inference*, which translates recursive programs with non-sized (co)inductive types to their sized counterparts when they are well-defined. Since our view is that sized types are a mode of use of more general arithmetic refinements, we do not consider size inference at the moment.

**Infinite and circular proofs.** Validity conditions of infinite proofs have been developed to keep cut elimination productive, which correspond to criteria like the guardedness check [30, 31, 7]. Although we use infinite typing derivations, we explicitly avoid syntactic termination checking for its non-compositionality. Nevertheless, we are interested in implementing such validity conditions as uses of sized types as future work. Relatedly, cyclic termination proofs for separation logic programs can be automated [14, 58], although it is unclear how they could generalize to concurrent programs (in the setting of concurrent separation logic) as well as codata.

**Session types.** Session types are inextricably linked with SAX, as it also has an asynchronous message passing interpretation [52]. Severi et al. [56] give a mixed functional and concurrent programming language where corecursive definitions are typed with Nakano's later modality [48]. Since Vezzosi [62] gives an embedding of the later modality and its dual into sized types, we believe that a similar arrangement can be achieved in our setting. In any case, we support recursion schemes more complex than structural (co)recursion [47].

**$\pi$-calculi.** Certain type systems for $\pi$-calculi [42, 49, 35] guarantee the eventual success of communication only if or regardless of whether processes diverge [23]. Considering a configuration $C$ such that $\Gamma \vdash C :: (\Gamma, a : X[n])$ where $X[i]$ is a positive coinductive type, we conjecture that $|C|$, which has all constraint and arithmetic data erased, is similarly "productive" even if it may *not* terminate. Intuitively, $C$ writes a number of cells as a function of $n$ then terminates, so $|C|$ represents $C$ in the limit since $X[i]$ is positive coinductive. However, this behavior is more desirable in a message passing setting rather than in our shared memory setting.

On the other hand, there are type systems that themselves guarantee termination – some assign numeric *levels* to each channel name and restrict communication such that a measure induced by said levels decreases consistently [29, 28, 20]. While message passing is a different setting than ours, we are interested in the relationship between sizes and levels, if any. Other such type systems constrain the type and/or term structure; the language $\mathscr{P}$ [55] requires grammatical restrictions on both types and terms, the latter of which we are trying to avoid. On the other hand, the combination of linearity and a certain acyclicity condition [67] on graph types [66] is also sufficient. Our system is able to guarantee termination despite utilizing non-linear types, but it remains open how type refinements compare to graph types.

## 6 Conclusion and Future Work

We have presented a highly general concurrent language that conceives mixed inductive-coinductive programming as a mode of use of arithmetic refinements. Moreover, we prove normalization via a novel logical relations argument in the presence of infinitely deep typing derivations that is mediated through infinitely wide but finitely deep (inductive) typing. There are three main points of interest for future work.

1. *Richer types*: to mix linear [50], affine linear, non-linear, etc. references to memory as well as persistent and ephemeral memory, we conjecture that moving to a type system based on adjoint logic [52] is appropriate. In that case, sizes could be related to the grades of the adjoint modalities [57]. Furthermore, we are interested in generalizing to substructural polymorphic, higher-kinded [24], and dependent types [16, 43].
2. *Implementation*: we are interested in developing a convenient surface language (perhaps a functional one [51]) for SAX and implementing our type system, following Rast [25], an implementation of resource-aware session types that includes arithmetic refinements. Perhaps various validity conditions of infinite proofs can be implemented as implicit uses of sized type refinements.
3. *Message passing*: we would like to transport our results to the asynchronous message passing interpretation of SAX [52], avoiding a technically difficult detour through asynchronous typed $\pi$-calculi [32].

## References


1 Andreas Abel. Productive infinite objects via copatterns and sized types in agda.

2 Andreas Abel. Semi-continuous Sized Types and Termination. *Logical Methods in Computer Science*, Volume 4, Issue 2, April 2008.

3 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.

4 Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016.

5 Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, page 247, USA, 2001. IEEE Computer Society.

6 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

7 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary Proof Theory: the Multiplicative Additive Case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comput. Sci.*, 14(1):97–141, 2004. doi:10.1017/S0960129503004122.

9 Henning Basold. *Mixed inductive-coinductive reasoning: types, programs and logic.* PhD thesis, Radboud University Nijmegen, April 2018.

10 Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, pages 24–39, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

11 Frédéric Blanqui and Colin Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 105–119, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 Frédéric Blanqui and Cody Roux. On the relation between sized-types based termination and semantic labelling. In *18th EACSL Annual Conference on Computer Science Logic - CSL 09*, Coimbra, Portugal, September 2009. Full version.

13 James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

14 James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

15 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

16 Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, page 264, USA, 1996. IEEE Computer Society.

17 Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, 2009. Special issue: 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

18 Jonathan Chan and William J. Bowman. Practical sized typing for coq. *CoRR*, abs/1912.05601, 2019. arXiv:1912.05601.

**19** Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2):261–300, 2001.

**20** Ioana Cristescu and Daniel Hirschkoff. Termination in a $\pi$-calculus with subtyping. *Mathematical Structures in Computer Science*, 26(8):1395–1432, 2016.

**21** Francesco Dagnino. Foundations of regular coinduction. *Logical Methods in Computer Science*, Volume 17, Issue 4, October 2021. `doi:10.46298/lmcs-17(4:2)2021`.

**22** Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction, 2009.

**23** Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *Journal of Logical and Algebraic Methods in Programming*, 124:100717, 2022.

**24** Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Subtyping on Nested Polymorphic Session Types, 2021. `arXiv:2103.15193`.

**25** Ankush Das and Frank Pfenning. Rast: A Language for Resource-Aware Session Types, 2020. `arXiv:2012.13129`.

**26** Ankush Das and Frank Pfenning. Session Types with Arithmetic Refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

**27** Ankush Das and Frank Pfenning. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20, New York, NY, USA, 2020. Association for Computing Machinery.

**28** Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 328–342, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

**29** Yuxin Deng and Davide Sangiorgi. Ensuring termination by typability. *Information and Computation*, 204(7):1045–1082, 2006.

**30** Farzaneh Derakhshan and Frank Pfenning. Circular Proofs as Session-Typed Processes: A Local Validity Condition. *CoRR*, abs/1908.01909, August 2019.

**31** Farzaneh Derakhshan and Frank Pfenning. Circular Proofs in First-Order Linear Logic with Least and Greatest Fixed Points. *CoRR*, abs/2001.05132, January 2020.

**32** Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 228–242, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

**33** Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-Axiomatic Sequent Calculus. 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 29:1–29:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

**34** Derek Dreyer, Amin Timany, Robbert Krebbers, Lars Birkedal, and Ralf Jung. What Type Soundness Theorem Do You Really Want to Prove?, October 2019.

**35** Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 63–77, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

**36** Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.

**37** Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

**38** Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In *Proceedings of the 10th ACM SIGPLAN Interna-*

*tional Conference on Certified Programs and Proofs*, CPP 2021, pages 178–198, New York, NY, USA, 2021. Association for Computing Machinery.

**39** John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 410–423, New York, NY, USA, 1996. Association for Computing Machinery.

**40** Aleksandar Ignjatovic. Hilbert's program and the omega-rule, June 2018.

**41** Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

**42** Naoki Kobayashi. A new type system for deadlock-free processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, pages 233–247, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

**43** Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 17–30, New York, NY, USA, 2015. Association for Computing Machinery.

**44** Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *SIGPLAN Not.*, 36(3):81–92, January 2001.

**45** Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1), February 2019.

**46** Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2).* Kluwer Academic Publishers, USA, 2004.

**47** Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. *SIGPLAN Not.*, 51(9):434–447, September 2016.

**48** Hiroshi Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

**49** Luca Padovani. Deadlock and lock freedom in the linear $\pi$-calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

**50** Frank Pfenning. Types and Programming Languages, 2020.

**51** Klaas Pruiksma and Frank Pfenning. Back to Futures. *CoRR*, abs/2002.04607, February 2020.

**52** Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming*, 120:100637, 2021.

**53** Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

**54** Jorge Luis Sacchini. Linear Sized Types in the Calculus of Constructions. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 169–185, Cham, 2014. Springer International Publishing.

**55** Davide Sangiorgi. Termination of processes. *Mathematical Structures in Computer Science*, 16(1):1–39, 2006.

**56** Paula Severi, Luca Padovani, Emilio Tuosto, and Mariangiola Dezani-Ciancaglini. On Sessions and Infinite Data. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages*, pages 245–261, Cham, 2016. Springer International Publishing.

**57** Siva Somayyajula. Towards Unifying (Co)induction and Structural Control. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*, Rome (virtual), Italy, June 2021.

**58** Gadi Tellez and James Brotherston. Automatically verifying temporal properties of pointer programs with cyclic proof. *Journal of Automated Reasoning*, 64(3):555–578, 2020. `doi:10.1007/s10817-019-09532-0`.

**59** The Coq Development Team. The Coq Proof Assistant, January 2021.

**60** René Thiemann and Jürgen Giesl. Size-change termination for term rewriting. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, pages 264–278, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

**61** Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 350–369, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

**62** Andrea Vezzosi. Total (Co)Programming with Guarded Recursion. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, pages 77–78, Tallinn, Estonia, 2015. Institute of Cybernetics at Tallinn University of Technology.

**63** 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.

**64** Hongwei Xi. Dependent types for program termination verification. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 231–242, 2001.

**65** Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

**66** Nobuko Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 371–386, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

**67** Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the $\pi$-calculus. *Information and Computation*, 191(2):145–202, 2004.


## A Appendix

▶ **Example 22** (Typechecking). The process definition below, whose type signature is $i; \cdot; x : \text{nat}[i] \vdash^i y \leftarrow \text{eat}\ \ i\ x :: (y : \mathbf{1})$, traverses a unary natural number by induction to produce a unit. Recall $\text{nat}[i] = \oplus\{\text{zero} : \mathbf{1}, \text{succ} : i > 0 \wedge \text{nat}[i-1]\}$.

$$y \leftarrow \text{eat}\ \ i\ x = \mathbf{case}\, x^{\mathrm{R}}\ \{\text{zero}\ z \Rightarrow y^{\mathrm{W}} \leftarrow z^{\mathrm{R}}, \text{succ}\ z \Rightarrow y \leftarrow \text{eat}\ (i-1)\ z\}$$

Now, let us construct a typing derivation of its body below.

$$D = \dfrac{\dfrac{}{z : \mathbf{1} \vdash^i (y : \mathbf{1})}\ \text{id} \qquad \dfrac{\dfrac{i; i > 0 \vdash i - 1 < i \qquad \dfrac{[(i-1)/i][z/x]D \in i; \cdot; z : \text{nat}[i-1] \vdash^{i-1} (y : \mathbf{1})}{i; \cdot; z : \text{nat}[i-1] \vdash^{i-1}_{\infty} (y : \mathbf{1})}\ \infty}{i; i > 0; z : \text{nat}[i-1] \vdash^i (y : \mathbf{1})}\ \text{call}}{i; \cdot; z : i > 0 \wedge \text{nat}[i-1] \vdash^i (y : \mathbf{1})}\ \wedge\text{L}}{i; \cdot; x : \text{nat}[i] \vdash^i (y : \mathbf{1})}\ \oplus\text{L}$$

For space, we omit the process terms. Of importance is the instance of the call rule for the recursive call to eat: the check $i - 1 < i$ (discharged automatically) verifies that the process terminates and the loop $[(i-1)/i][z/x]D$ "ties the knot" on the typechecking process (the constraint $i > 0$ is implicitly weakened). Mutually recursive programs, then, are checked by circular typing derivations that are mutually recursive *in the metatheory*.

$$\frac{}{\Gamma, x : A \vdash^\omega y^{\mathrm{W}} \leftarrow x^{\mathrm{R}} :: (y : A)}\ \mathrm{id}^\omega \qquad \frac{\Gamma \vdash^\omega P(x) :: (x : A) \quad \Gamma, x : A \vdash^\omega Q(x) :: (z : C)}{\Gamma \vdash^\omega x \leftarrow P(x); Q(x) :: (z : C)}\ \mathrm{cut}^\omega$$

$$\frac{}{\Gamma \vdash^\omega x^{\mathrm{W}}.\langle\rangle :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R}^\omega \qquad \frac{\Gamma, x : \mathbf{1} \vdash^\omega P :: (z : C)}{\Gamma, x : \mathbf{1} \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle\rangle \Rightarrow P) :: (z : C)}\ \mathbf{1}\mathrm{L}^\omega$$

$$\frac{}{\Gamma, y : A, z : B \vdash^\omega x^{\mathrm{W}}.\langle y, z\rangle :: (x : A \otimes B)}\ \otimes\mathrm{R}^\omega \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \vdash^\omega P(y, z) :: (w : C)}{\Gamma, x : A \otimes B \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle y, z\rangle \Rightarrow P(y, z)) :: (w : C)}\ \otimes\mathrm{L}^\omega$$

$$\frac{\Gamma, y : A \vdash^\omega P(y, z) :: (z : B)}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle y, z\rangle \Rightarrow P(y, z)) :: (x : A \to B)}\ \to\mathrm{R}^\omega \qquad \frac{}{\Gamma, x : A \to B, y : A \vdash^\omega x^{\mathrm{R}}.\langle y, z\rangle :: (z : B)}\ \to\mathrm{L}^\omega$$

$$\frac{k \in S}{\Gamma, y : A_k \vdash^\omega x^{\mathrm{W}}.k\, y :: (x : \oplus\{\ell : A_\ell\}_{\ell \in S})}\ \oplus\mathrm{R}^\omega \qquad \frac{\{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S}, y : A_\ell \vdash^\omega P(y) :: (z : C)\}_{\ell \in S}}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in S} \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (z : C)}\ \oplus\mathrm{L}^\omega$$

$$\frac{\{\Gamma \vdash^\omega P(y) :: (y : A_\ell)\}_{\ell \in S}}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, \{\ell\, y \Rightarrow P_\ell(y)\}_{\ell \in S} :: (x : \&\{\ell : A_\ell\}_{\ell \in S})}\ \&\mathrm{R}^\omega \qquad \frac{k \in S}{\Gamma, x : \&\{\ell : A_\ell\}_{\ell \in S} \vdash^\omega x^{\mathrm{R}}.k\, y :: (y : A_k)}\ \&\mathrm{L}^\omega$$

$$\frac{}{\Gamma, y : A(n) \vdash^\omega x^{\mathrm{W}}.\langle n, y\rangle :: (x : \exists i.\, A(n))}\ \exists\mathrm{R}^\omega \qquad \frac{\Gamma, x : \exists i.\, A(i), y : A(n) \vdash^\omega P(n, y) :: (z : C) \text{ for all } n \in \mathbb{N}}{\Gamma, x : \exists i.\, A(i) \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle i, y\rangle \Rightarrow P(i, y)) :: (z : C)}\ \exists\mathrm{L}^\omega$$

$$\frac{\Gamma \vdash^\omega P(n, y) :: (y : A(n)) \text{ for all } n \in \mathbb{N}}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle i, y\rangle \Rightarrow P(i, y)) :: (x : \forall i.\, A(i))}\ \forall\mathrm{R}^\omega \qquad \frac{}{\Gamma, x : \forall i.\, A(i) \vdash^\omega x^{\mathrm{R}}.\langle n, y\rangle :: (y : A(n))}\ \forall\mathrm{L}^\omega$$

$$\frac{\cdot ; \cdot \vdash \phi}{\Gamma, y : A \vdash^\omega x^{\mathrm{W}}.\langle *, y\rangle :: (x : \phi \wedge A)}\ \wedge\mathrm{R}^\omega \qquad \frac{\Gamma, x : \phi \wedge A, y : A \vdash^\omega P(y) :: (z : C) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \wedge A \vdash^\omega \mathbf{case}\, x^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (z : C)}\ \wedge\mathrm{L}^\omega$$

$$\frac{\Gamma \vdash^\omega P(y) :: (y : A) \text{ if } \cdot ; \cdot \vdash \phi}{\Gamma \vdash^\omega \mathbf{case}\, x^{\mathrm{W}}\, (\langle *, y\rangle \Rightarrow P(y)) :: (x : \phi \Rightarrow A)}\ \Rightarrow\mathrm{R}^\omega \qquad \frac{\cdot ; \cdot \vdash \phi}{\Gamma, x : \phi \Rightarrow A \vdash^\omega x^{\mathrm{R}}.\langle *, y\rangle :: (y : A)}\ \Rightarrow\mathrm{L}^\omega$$

$$\text{(no rule for } \mathbf{impossible}\text{)} \qquad \frac{y \leftarrow f\ \bar{i}\ \overline{x} = P_f(\bar{i}, \overline{x}, y) \quad \overline{x} : \overline{A} \vdash^\omega P_f(\overline{n}, \overline{x}, y) :: (y : A)}{\Gamma, \overline{x} : \overline{A} \vdash^\omega y \leftarrow f\ \overline{n}\ \overline{x} :: (y : A)}\ \mathrm{call}^\omega$$

**Figure 5** SAX$^\omega$ Typing Rules.

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A} \vDash \text{proc}\, b\, (b^{\mathrm{W}} \leftarrow a^{\mathrm{R}}) :: (b : \mathscr{A})}\ \text{id} \qquad \frac{\mathbf{\Gamma} \vDash \text{proc}\, a\, P :: (a : \mathscr{A}) \quad \mathbf{\Gamma}, a : A \vDash \text{proc}\, c\, (Q(a)) :: (c : \mathscr{C})}{\mathbf{\Gamma} \vDash \text{proc}\, c\, (x \leftarrow P; Q(x)) :: (c : \mathscr{C})}\ \text{cut}$$

$$\frac{}{\mathbf{\Gamma} \vDash \text{!cell}\, a\, \langle\rangle :: (a : \mathbb{1})}\ \mathbb{1}\text{R} \qquad \frac{\mathbf{\Gamma}, a : \mathbb{1} \vDash \text{proc}\, b\, P :: (b : \mathscr{C})}{\mathbf{\Gamma}, a : \mathbb{1} \vDash \text{proc}\, b\, (\mathbf{case}\, a^{\mathrm{R}}\, (\langle\rangle \Rightarrow P)) :: (b : \mathscr{C})}\ \mathbb{1}\text{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A}, b : \mathscr{B} \vDash \text{!cell}\, c\, \langle a, b\rangle :: (c : \mathscr{A} \otimes \mathscr{B})}\ \otimes\text{R} \qquad \frac{\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B}, a : \mathscr{A}, b : \mathscr{B} \vdash \text{proc}\, d\, (P(a,b)) :: (d : \mathscr{D})}{\mathbf{\Gamma}, c : \mathscr{A} \otimes \mathscr{B} \vdash \text{proc}\, d\, (\mathbf{case}\, c^{\mathrm{R}}\, (\langle x, y\rangle \Rightarrow P(x,y))) :: (d : \mathscr{D})}\ \otimes\text{L}$$

$$\frac{\mathbf{\Gamma}, a : \mathscr{A} \vDash \text{proc}\, b\, (P(a,b)) :: (b : \mathscr{B})}{\mathbf{\Gamma} \vDash \text{!cell}\, c\, (\langle x, y\rangle \Rightarrow P(x,y)) :: (c : \mathscr{A} \to \mathscr{B})}\ \to\text{R} \qquad \frac{}{\mathbf{\Gamma}, c : \mathscr{A} \to \mathscr{B}, a : \mathscr{A} \vDash \text{proc}\, b\, (c^{\mathrm{R}}.\langle a, b\rangle) :: (b : \mathscr{B})}\ \to\text{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{A}_k \vDash \text{!cell}\, b\, (k\, a) :: (b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S})}\ \oplus\text{R} \qquad \frac{\{\mathbf{\Gamma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S}, a : \mathscr{A}_k \vDash \text{proc}\, c\, (P_k(a)) :: (c : \mathscr{C})\}_{k \in S}}{\mathbf{\Gamma}, b : \oplus\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \text{proc}\, c\, (\mathbf{case}\, b^{\mathrm{R}}\, \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S}) :: (c : \mathscr{C})}\ \oplus\text{L}$$

$$\frac{\{\mathbf{\Gamma} \vDash \text{proc}\, a\, (P_\ell(a)) :: (a : \mathscr{A}_\ell)\}_{\ell \in S}}{\mathbf{\Gamma} \vDash \text{!cell}\, b\, \{\ell\, x \Rightarrow P_\ell(x)\}_{\ell \in S} :: (b : \&\{\ell : \mathscr{A}_\ell\}_{\ell \in S})}\ \&\text{R} \qquad \frac{}{\mathbf{\Gamma}, b : \&\{\ell : \mathscr{A}_\ell\}_{\ell \in S} \vDash \text{proc}\, b\, (c^{\mathrm{R}}.k\, a) :: (a : \mathscr{A})}\ \&\text{L}$$

$$\frac{}{\mathbf{\Gamma}, a : \mathscr{F}(n) \vDash \text{!cell}\, b\, \langle n, a\rangle :: (b : \exists \mathscr{F})}\ \exists\text{R} \qquad \frac{\{\mathbf{\Gamma}, b : \exists \mathscr{F}, a : \mathscr{F}(n) \vDash \text{proc}\, c\, (P(n,a)) :: (c : \mathscr{C})\}_{n \in \mathbb{N}}}{\mathbf{\Gamma}, b : \exists \mathscr{F} \vDash \text{proc}\, c\, (\mathbf{case}\, b^{\mathrm{R}}\, (\langle i, x\rangle \Rightarrow P(i,x))) :: (c : \mathscr{C})}\ \exists\text{L}$$

$$\frac{\{\mathbf{\Gamma} \vDash \text{proc}\, a\, (P(n,a)) :: (a : \mathscr{F}(n))\}_{n \in \mathbb{N}}}{\mathbf{\Gamma} \vDash \text{!cell}\, b\, (\langle i, x\rangle \Rightarrow P(i,x)) :: (b : \forall \mathscr{F})}\ \forall\text{R} \qquad \frac{}{\mathbf{\Gamma}, b : \forall \mathscr{F} \vDash \text{proc}\, a\, (b^{\mathrm{R}}.\langle n, a\rangle) :: (a : \mathscr{F}(n))}\ \forall\text{L}$$

$$\frac{\cdot ; \cdot \vdash \phi}{\mathbf{\Gamma}, b : A \vDash \text{!cell}\, a\, \langle *, b\rangle :: (a : \phi \wedge \mathscr{A})}\ \wedge\text{R} \qquad \frac{\mathbf{\Gamma}, a : \phi \wedge \mathscr{A}, b : \mathscr{A} \vdash \text{proc}\, a\, (P(b)) :: (c : \mathscr{C}) \quad \text{if } \cdot ; \cdot \vdash \phi}{\mathbf{\Gamma}, a : \phi \wedge \mathscr{A} \vDash \text{proc}\, a\, (\mathbf{case}\, a^{\mathrm{R}}\, (\langle *, y\rangle \Rightarrow P(y))) :: (c : \mathscr{C})}\ \wedge\text{L}$$

$$\frac{\mathbf{\Gamma} \vDash \text{proc}\, b\, (P(b)) :: (b : \mathscr{A}) \text{ if } \cdot ; \cdot \vdash \phi}{\mathbf{\Gamma} \vDash \text{!cell}\, a\, (\langle *, y\rangle \Rightarrow P(y)) :: (a : \phi \Rightarrow \mathscr{A})}\ \Rightarrow\text{R} \qquad \frac{\cdot ; \cdot \vdash \phi}{\mathbf{\Gamma}, a : \phi \Rightarrow \mathscr{A} \vDash \text{proc}\, b\, (a^{\mathrm{R}}.\langle *, b\rangle) :: (b : \mathscr{A})}\ \Rightarrow\text{L}$$

$$\text{(no rule for } \mathbf{impossible}\text{)} \qquad \frac{y \leftarrow f\ \overline{i}\ \overline{x} = P_f(\overline{i}, \overline{x}, y) \quad \overline{b} : \overline{\mathscr{A}} \vDash \text{proc}\, a\, (P_f(\overline{n}, \overline{b}, a)) :: (a : \mathscr{A})}{\mathbf{\Gamma}, \overline{b} : \overline{\mathscr{A}} \vDash \text{proc}\, a\, (a \leftarrow f\ \overline{n}\ \overline{b}) :: (a : \mathscr{A})}\ \text{call}$$

**Figure 6** Semantic Object Typing Rules.

# Addition and Differentiation of ZX-Diagrams

**Emmanuel Jeandel**
LORIA, CNRS, Université de Lorraine, Inria Mocqua, Nancy, France

**Simon Perdrix**
LORIA, CNRS, Université de Lorraine, Inria Mocqua, Nancy, France

**Margarita Veshchezerova**
LORIA, CNRS, Université de Lorraine, Inria Mocqua, Nancy, France
EDF R&D, France

## Abstract

The ZX-calculus is a powerful framework for reasoning in quantum computing. It provides in particular a compact representation of matrices of interests. A peculiar property of the ZX-calculus is the absence of a formal sum allowing the linear combinations of arbitrary ZX-diagrams. The universality of the formalism guarantees however that for any two ZX-diagrams, the sum of their interpretations can be represented by a ZX-diagram. We introduce a general, inductive definition of the addition of ZX-diagrams, relying on the construction of controlled diagrams. Based on this addition technique, we provide an inductive differentiation of ZX-diagrams.

Indeed, given a ZX-diagram with variables in the description of its angles, one can differentiate the diagram according to one of these variables. Differentiation is ubiquitous in quantum mechanics and quantum computing (e.g. for solving optimization problems). Technically, differentiation of ZX-diagrams is strongly related to summation as witnessed by the product rules.

We also introduce an alternative, non inductive, differentiation technique rather based on the isolation of the variables. Finally, we apply our results to deduce a diagram for an Ising Hamiltonian.

**2012 ACM Subject Classification** Theory of computation → Quantum computation theory; Theory of computation → Axiomatic semantics

**Keywords and phrases** ZX calculus, Addition of ZX diagrams, Diagrammatic differentiation

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.13

**Related Version** *Full Version*: `https://arxiv.org/abs/2202.11386` [12]

**Funding** This work was supported in part by the CIFRE EDF/Loria Quantum Computing for Combinatorial Optimisation, the French National Research Agency (ANR) under the research projects SoftQPro ANR-17-CE25-0009-02 and VanQuTe ANR-17-CE24-0035, by the DGE of the French Ministry of Industry under the research project PIA-GDN/QuantEx P163746-484124, by the PEPR integrated project EPiQ, by the STIC-AmSud project Qapla' 21-STIC-10, and by the European projects NEASQC (funded from the European Union's Horizon 2020 research and innovation programme grant agreement No 951821) and HPCQS (European High-Performance Computing Joint Undertaking under grant agreement No 101018180).

**Acknowledgements** The authors want to thank Bob Coecke, Harny Wang, and Richie Yeung for their availability and the fruitful discussions in the last few days prior to the submission.

## 1 Introduction

The ZX-calculus is a graphical language for manipulating linear maps. It was originally introduced in [4] and proven to be complete for qubit quantum computation [13, 11, 16, 25]. A general introduction to the language alongside the overview of the main applications is available in [24].

© Emmanuel Jeandel, Simon Perdrix, and Margarita Veshchezerova;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 13; pp. 13:1–13:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Due to its flexibility, ZX-calculus became widely used to address different problems of quantum computing. However, its application to the rapidly growing field of variational algorithms [3] like QAOA [7] *(quantum approximation optimization algorithm)* and VQE [20] (*variational quantum eigensolver*) are so far limited. Nevertheless as variational algorithms do not require heavy resource for error-correction, the incoming emergence of *NISQ* devices makes from them an object of particular attention [21]. We believe that the reason why they are still unexplored with the means of ZX-calculus is the absence of a convenient way to differentiate parametrized diagrams. Indeed, basic building blocks of variational algorithms are parametrized circuits and the search of optimal parameter values is a crucial part of these algorithms. The search is usually done by classical numerical optimization methods [8] and most of them use derivatives.

The main difficulty for differentiation of ZX-diagrams comes from the *product rules* that involve sums. Several attempts were made to face this problem [29, 23]. The paper [23] extends the signature of ZX-category to formal sums of diagrams while [29] provides explicit derivatives for diagrams with the number of parameter occurrences limited to two. The first option that is to use formal sums has major disadvantages as there is no rules to manipulate sums of ZX-diagrams.

In our approach the derivative of a parametrized ZX-diagram is another ZX-diagram. Hence we avoid the extension of the signature with formal sums. In order to tackle sums that appear in the product rule, we introduce an original technique to perform the addition of diagrams entirely in the ZX-calculus. We use special diagrams called controlled states [15]. We suggest a way to represent every ZX-diagram by such a state. As we know how to sum controlled states [15] the addition for arbitrary diagrams follows. An inductive definition of the derivative is obtained by explicit diagrammatic representation of the product rules.

Very recently an independent work with a similar result, although obtained from a different approach, was published on *arXiv* [27]. In contrast to our work, the authors of [27] use algebraic ZX-calculus [26] and W-spiders [10] to express derivatives. Their paper highlights the crucial role that W-spider plays in the representation of sums. However, it does not provide an algorithm of diagrammatic addition for arbitrary diagrams.

In an attempt to give a ready-to-use toolbox for differentiation, we provide an easy and convenient way to compute the derivative for the family of linear diagrams ZX($\beta$) [15]. Most of circuits for variational algorithms belong to ZX($\beta$) and we believe that our formulas will make the analysis of them much simpler.

In the end, we show how our result together with the Stone's theorem [22] allows to find a ZX-diagram for an Ising Hamiltonian - another key component of variational quantum algorithms [9].

## Structure of the paper

In the section 2, we give a brief introduction to the ZX-calculus. In the section 3, we recall the properties of controlled states and give the definition of *controlizer*: a map that transforms an arbitrary diagram to a controlled state. We show how to use controlizers to perform the addition of ZX-diagrams. In the section 4, we introduce the formal semantics of derivative of a parametrized diagram. The definition is followed by an algorithm for differentiation that explicitly incorporates the product rule. Finally, we give two compact formulas for derivatives in ZX($\beta$) that may be directly used in computation. In the section 5, we show how to apply our result to obtain a diagram for an Ising Hamiltonian. Most of proofs are detailed in the full version of our paper that is available online [12].

## 2 ZX-calculus

### 2.1 Syntax and Semantics

The ZX-diagrams are generated by green spiders, red spiders and Hadamard, where both kinds of spiders have an arbitrary number of inputs/outputs and are decorated with angles. ZX-diagrams are also made of wires: the identity, the swap and also the possibility to bend wires with a cup and a cap. Finally, the empty diagram is denoted.

▶ **Definition 1.** ZX-*diagrams are inductively defined as follows: for* $n, m \in \mathbb{N}$ *and* $\alpha \in \mathbb{R}/2\pi\mathbb{Z}$,

(green spider $\alpha$, $n$ inputs, $m$ outputs) : $n \to m$ (red spider $\alpha$, $n$ inputs, $m$ outputs) : $n \to m$ (Hadamard) : $1 \to 1$ (identity) : $1 \to 1$

(cup) : $2 \to 0$ (cap) : $0 \to 2$ (swap) : $2 \to 2$ (empty) : $0 \to 0$

*are* ZX-*diagrams, and for any* ZX-*diagrams* $D_0 : a \to b$, $D_1 : b \to c$, *and* $D_2 : c \to d$, $D_1 \circ D_0 : a \to c$ *and* $D_0 \otimes D_2 : a + c \to b + d$ *are* ZX-*diagrams. Pictorially:*

$D_1 \circ D_0$ = ($D_0$ stacked above $D_1$) *and* $D_0 \otimes D_2$ = ($D_0$ beside $D_2$)

A diagram with no input/ouput is called a *scalar*. In order to compactly write scalar factors, we introduce syntactic sugar $[-]^{\otimes n}$. For any scalar $d : 0 \to 0$ the notation $d^{\otimes n}$ corresponds to $\underbrace{d \otimes \cdots \otimes d}_{n}$.

Semantically, ZX-diagrams are standardly interpreted as linear maps, and thus they can be used to represent quantum evolutions.

▶ **Definition 2.** *For any* ZX-*diagram* $D : n \to m$, *let* $[\![D]\!] \in \mathcal{M}_{2^m,2^n}(\mathbb{C})$ *be inductively defined as:* $[\![D_1 \circ D_0]\!] = [\![D_1]\!] \circ [\![D_0]\!]$, $[\![D_0 \otimes D_2]\!] = [\![D_0]\!] \otimes [\![D_2]\!]$, *and*

$$[\![\text{green spider } \alpha]\!] = |0\rangle^{\otimes m}\langle 0|^{\otimes n} + e^{i\alpha}|1\rangle^{\otimes m}\langle 1|^{\otimes n}, \quad [\![\text{red spider } \alpha]\!] = |+\rangle^{\otimes m}\langle +|^{\otimes n} + e^{i\alpha}|-\rangle^{\otimes m}\langle -|^{\otimes n}$$

$$[\![\,|\,]\!] = |0\rangle\langle 0| + |1\rangle\langle 1|, \quad [\![\text{Hadamard}]\!] = |+\rangle\langle 0| + |-\rangle\langle 1|, \quad [\![\text{empty}]\!] = 1$$

$$[\![\text{cup}]\!] = \langle 00| + \langle 11|, \quad [\![\text{cap}]\!] = |00\rangle + |11\rangle, \quad [\![\text{swap}]\!] = \sum_{i,j\in\{0,1\}} |ij\rangle\langle ji|$$

*where* bra-ket *notations are used:* $|0\rangle = \binom{1}{0}$, $|1\rangle = \binom{0}{1}$, $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$, $|-\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$, $|xy\rangle = |x\rangle \otimes |y\rangle$ *and* $\langle x| = |x\rangle^\dagger$ *is the adjoint (complex conjugate) of* $|x\rangle$.

Sometimes it is meaningful to consider diagrams with angles from a restricted sub-group $\mathcal{G}$ of $\mathbb{R}/2\pi\mathbb{Z}$. Such restrictions lead to *fragments* of the language, denoted $\text{ZX}_\mathcal{G}$-calculus [15]. The standard interpretation associates to each $\text{ZX}_\mathcal{G}$-diagram $D : n \to m$ a matrix $[\![D]\!] \in \mathcal{M}_{2^m,2^n}(\mathcal{R}_\mathcal{G})$ with elements in the ring $\mathcal{R}_\mathcal{G} = \mathbb{Z}\left[\frac{1}{\sqrt{2}}, e^{i\mathcal{G}}\right]$ - the smallest ring that contains $\mathbb{Z}$, $\frac{1}{\sqrt{2}}$ and $\{e^{ia} | a \in \mathcal{G}\}$ [15].

In particular the $\frac{\pi}{2}$- (resp. $\pi$-) fragment[1], also called Clifford (resp. real Clifford) fragment, enjoys nice properties [1, 6] but is not universal for quantum computing, even approximately. Furthermore any quantum computation that can be expressed in this fragment can be efficiently simulated on a classical computer. As soon as the group contains the angle $\frac{\pi}{4}$, the corresponding fragment is approximatively universal for quantum computing: any $2^n \times 2^n$ unitary transformation can be approximated by a ZX-diagram from this fragment with arbitrary precision. In particular the $\frac{\pi}{4}$-fragment, also called "Clifford+T" fragment has been extensively studied [13, 17, 19]. Other finitely generated fragments have been considered in [15].

Notice that for any sub-group $\mathcal{G}$ of $\mathbb{R}/2\pi\mathbb{Z}$ that contains $\frac{\pi}{4}$, ZX$_\mathcal{G}$-diagrams are *universal* [15] in the sense that for any matrix $M \in \mathcal{M}_{2^m,2^n}(\mathcal{R}_\mathcal{G})$ there exists a ZX$_\mathcal{G}$-diagram $D : n \to m$ such that $[\![D]\!] = M$.

In this work we extensively use triangle: $\blacktriangle$ - a syntactic sugar introduced in [13]. It corresponds to a non-unitary transformation: $[\![\blacktriangle]\!] = |0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 1|$. The triangle may be written in terms of red and green spiders as:

(1)

## 2.2 The calculus

Two ZX-diagrams may have the same interpretation, as a consequence the language is equipped with a set of rewrite rules (Figure 1) that allows to transform diagrams.

In addition, ZX-diagrams can be deformed at will: all wires may be bent in any manner that keeps intact the order of inputs and outputs. It is also allowed to arbitrary change the order of wires for green and red spiders and the Hadamard. Corresponding transformation rules are aggregated under the paradigm *Only topology matters*:

We denote ZX $\vdash D_1 = D_2$ if $D_1$ may be transformed to $D_2$ by local application of rewriting rules.

The ZX-calculus is sound, i.e. the rules preserve the semantics: if ZX $\vdash D_1 = D_2$ then $[\![D_1]\!] = [\![D_2]\!]$. The converse property is called completeness. The set of rules (1) was proven complete for the $\frac{\pi}{4}$-fragment [15], and a single extra-rule makes the language complete for arbitrary diagrams [14]. Notice that alternative sets of rules have been shown to be complete for general ZX-diagrams [11, 25]. We choose to consider the rules of Figure 1 as they have been used to study diagrams with parameters in [14], which is an appropriate framework for differentiation (see section 4).

[1] I.e. the fragment of diagrams which angles are in the group generated by $\frac{\pi}{2}$ (resp. $\pi$)

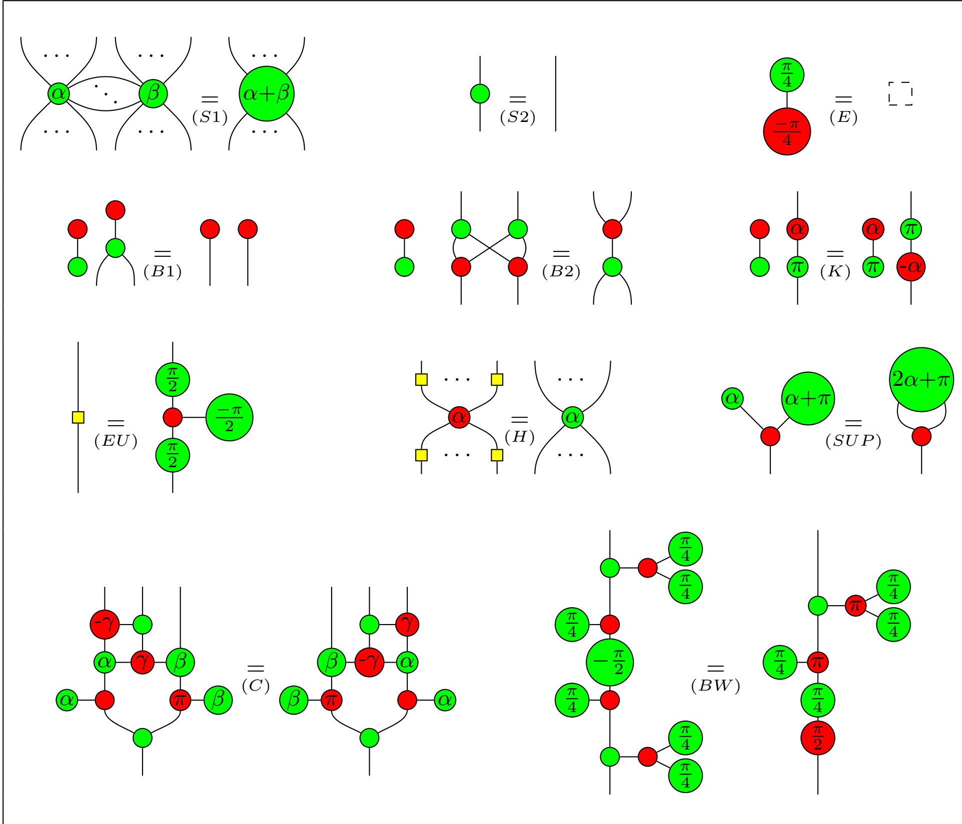



**Figure 1** Axioms for ZX as presented in [15]. All rules stay true flipped upside down and with inverted colors. Families of equations are given using "dots": $\dots$ means any number of wires, $\ddots$ means at least one wire.

## 3 Addition of ZX-diagrams

The ZX-calculus is a convenient tool for manipulating compositions and tensor products of linear maps. These two operations have natural physical interpretations, corresponding to sequential and parallel compositions respectively. The addition is a natural operation on matrices and it can be interpreted as the superposition phenomenon in quantum mechanics. However, the addition is not a physical process, hence it is not reflected in the standard ZX-calculus [5]. On the other hand, for any two diagrams $D_1, D_2 : n \to m$, the universality of the ZX-calculus guarantees that there exists a diagram $D : n \to m$ such that $[\![D]\!] = [\![D_1]\!] + [\![D_2]\!]$.

We provide in this section a general construction for such a diagram. As pointed out in [15] for the definition of normal forms in the ZX-calculus, one can inductively define the addition on "controlled" versions of the diagrams. A controlled version of a diagram $D_0$ is roughly speaking a diagram with an extra input such that when this extra input is set to $|1\rangle$ the diagram behaves as $D_0$ and when it is set to $|0\rangle$ the diagram behaves as a neutral diagram. In order to construct controlled versions, we pass by controlled states:

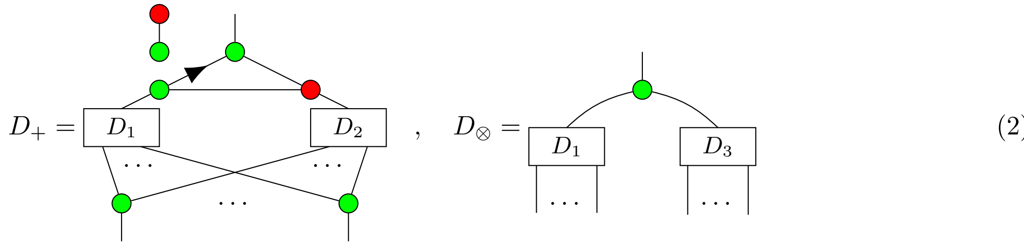
▶ **Definition 3** (Controlled state [15]). *A ZX-diagram $D : 1 \to n$ is a controlled state if* $[\![D]\!]\,|0\rangle = \sum_{x\in\{0,1\}^n} |x\rangle = \left[\!\!\left[\underbrace{\bullet \dots \bullet}_{n}\right]\!\!\right]$.

▶ **Example 4.** The diagram [diagram] is a controlled state for the scalar 0. Indeed, $[\![ \cdot ]\!]^{\otimes 2} \underset{35}{=} [\![\ ]\!] = 1$ and $[\![ \cdot ]\!]^{\otimes 2} = [\![ \cdot ]\!]^{\otimes 2} \times [\![\pi]\!] = [\![ \cdot ]\!]^{\otimes 2} \times (1 + e^{i\pi}) = 0$

Intuitively, a controlled state is a way to encode the state $[\![D]\!]\,|1\rangle$.

Controlled states have nice properties that allows to perform element-wise addition and tensor product of corresponding vectors:

▶ **Lemma 5** (Sum and tensor product [15]). *For any controlled states $D_1, D_2 : 1 \to n$ and $D_3 : 1 \to m$ the diagrams:*




(2)

*are controlled states, $[\![D_+]\!]\,|1\rangle = [\![D_1]\!]\,|1\rangle + [\![D_2]\!]\,|1\rangle$ and $[\![D_\otimes]\!]\,|1\rangle = [\![D_1]\!]\,|1\rangle \otimes [\![D_3]\!]\,|1\rangle$.*

Lemma 5 provides a way to obtain a sum of two diagrams in a controlled state form. In order to extend the addition to *arbitrary* diagrams we introduce *controlizers* - maps that associate diagrams with the corresponding controlled states. Formally,

▶ **Definition 6** (Controlizer). *We say that a map $C : \mathrm{ZX}(n,m) \to \mathrm{ZX}(1, n{+}m)$ that associates to every diagram $D : n \to m$ a diagram $C(D) : 1 \to n + m$ is controlizer if the following conditions hold for any ZX-diagram $D$:*

**(i)** *$C(D)$ is a controlled state*

**(ii)**

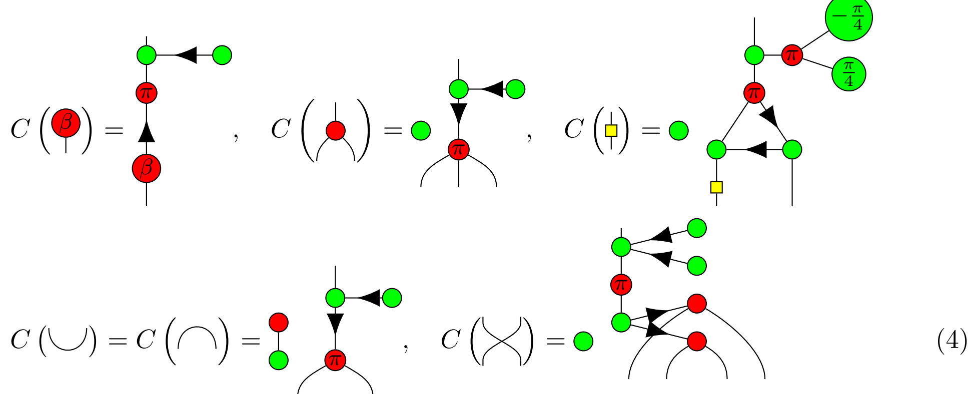
$$[\![D]\!] = [\![ \cdots C(D) \cdots ]\!] \quad (3)$$

(diagram with $\otimes n + m$, $n$, $\pi$, $C(D)$, $m$)

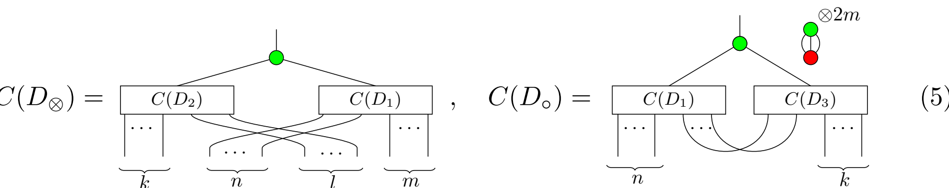
In this definition (and what follows) $\mathrm{ZX}(n,m)$ denotes the set of ZX-diagrams with $n$ inputs and $m$ outputs. If $n$ and $m$ are not specified, they may take arbitrary values.

▶ **Example 7** (Inductive controlizer). We define the map $C : \mathrm{ZX}(n,m) \to \mathrm{ZX}(1, n+m)$ that associates to each diagram $D : n \to m$ a diagram $C(D) : 1 \to n + m$:

**(i)** For the generators [red spider β], [red spider], [yellow box], ∩, ∪, [swap]:




**(ii)** Generators [red spider α] and [green spider α] can be decomposed as follows using the above generators:



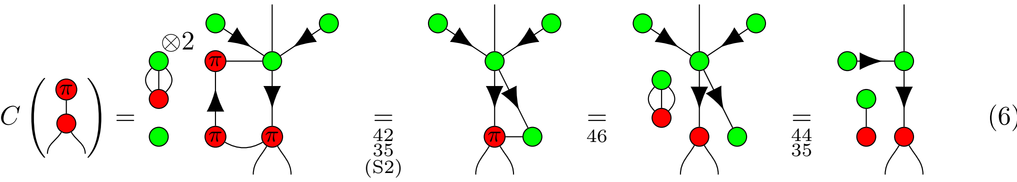

**(iii)** For tensor product $D_\otimes = D_2 \otimes D_1$ and composition $D_\circ = D_3 \circ D_1$ where $D_1 : n \to m$, $D_2 : k \to l$ and $D_3 : m \to k$:

$$C(D_\otimes) = \quad , \quad C(D_\circ) = \qquad (5)$$

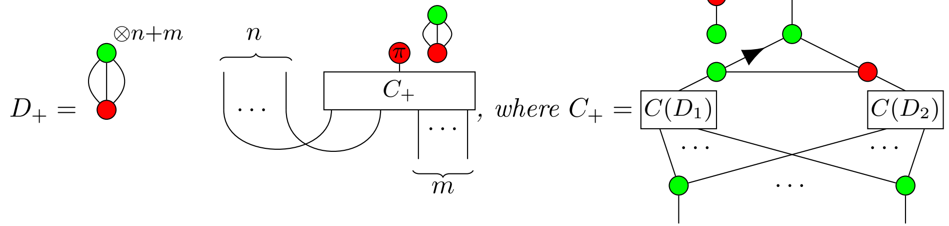
▶ **Lemma 8.** *The map from Example 7 satisfies the definition of controlizer.*

▶ Remark 9. A step-by-step application of the map $C$ may lead to different diagrams depending on the order of decomposition on tensor products and compositions. However, all possible outputs are semantically equivalent and by completeness of ZX-calculus are equivalent as diagrams.

▶ **Example 10.** We show how to obtain $C$ ([red π spider]) using definition 7:




▶ **Theorem 11.** *For diagrams $D_1 : n \to m$ and $D_2 : n \to m$ the diagram*

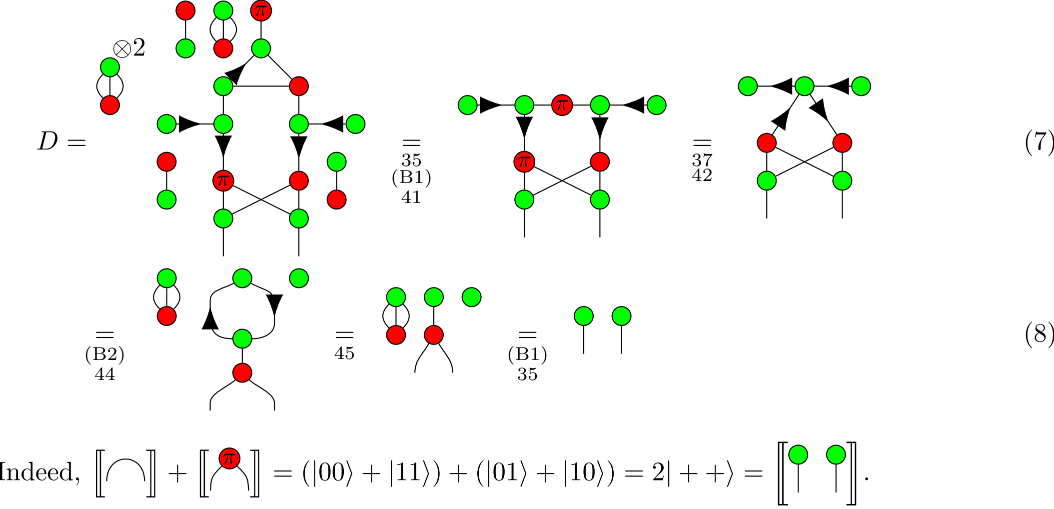



*is such that* $[\![D_+]\!] = [\![D_1]\!] + [\![D_2]\!]$.

**Proof (Theorem 11).** The theorem follows from the definition of controlizer and Lemma 5. ◀

We illustrate the diagrammatic addition with a simple example:

▶ **Example 12.** Using Theorem 11, we construct a diagram $D$ as the addition of and , which can be simplified as follows, using the rules of the ZX-calculus:




Indeed, $[\![\ ]\!] + [\![\ ]\!] = (|00\rangle + |11\rangle) + (|01\rangle + |10\rangle) = 2|++\rangle = [\![\ ]\!]$.

## 4 Differentiation of ZX-diagrams

Mathematically, ZX-diagrams form a symmetric monoidal category [4] with natural numbers as objects and diagrams as morphisms. Notice that a definition of the differential category *with respect to morphism's domain* is given in [2]. In the current work, in contrast, we operate parametrized morphisms and derivatives are considered *with respect to parameters*. For example, for the category of matrices with elements that are smooth functions on some variable $\beta$ we are interested in the derivative over $\beta$. We say that a ZX-diagram $D$ is parametrized by $\beta_1, \ldots, \beta_k$ if its angles are some functions on $\beta_1, \ldots, \beta_k$. We denote such a diagram by $D(\beta_1, \ldots, \beta_k)$. [2]

We want to define the formal semantics for the derivative of a parametrized diagram that is consistent with existing definitions of derivatives in monoidal categories with parametrized morphisms.

The work [23] defines the derivative for *monoidal categories with sum* [3] in the following way.

▶ **Definition 13** (Derivative [23]). *A derivative $\partial_M : C(x, y) \to C(x, y)$ in a monoidal category $M$ with sum $(+)$ is a sum-preserving unary operator that satisfies the following axioms (product rules):*

***$\circ$-product rule:*** $\partial_M[A \circ B] = \partial_M[A] \circ B + A \circ \partial_M[B]$

***$\otimes$-product rule:*** $\partial_M[A \otimes B] = \partial_M[A] \otimes B + A \otimes \partial_M[B]$

---

[2] We can *evaluate* each parametrized diagram $D(\boldsymbol{\beta})$, $\boldsymbol{\beta} \in \mathbb{R}^k$ in a point $\boldsymbol{\beta}^0 \in \mathbb{R}^k$ by replacing every occurrence of $\beta_i$ with the respective value $\beta_i^0$. The result of evaluation is a diagram $D(\boldsymbol{\beta}_0)$ from $ZX_{\mathbb{R}}$.

[3] Formally, sum $(+)$ is a commutative monoid that maps each pair of morphisms with same domains/codomains to another morphism. The sum is distributive with respect to composition and tensor product.

Because of the sums involved in product rules, we avoid to directly use the derivative as defined above. Indeed, even if the Theorem 11 provides a fully diagrammatic way for the addition of ZX-diagrams, the formal introduction of a sum monoid leads to unnecessary complications.

On the other hand, for any group $\mathcal{G}$ the category $\mathcal{M}(\mathcal{G})$ of matrices with elements in $\mathcal{G}$ admits a natural definition of sums: the sum $(+_M)$ of two matrices is obtained by entrywise addition. Therefore, we get the semantics of the derivative $\partial_M$ in $\mathcal{M}(\mathcal{G})$ directly from Definition 13. It was proven in [23] that for the category of parametrized linear maps with elements that are smooth functions $S : \mathbb{R}^n \to \mathbb{C}$ an entrywise differentiation of matrix elements satisfies both axioms.

Taking previous remarks in consideration, we suggest an alternative semantics for the derivative in the ZX-calculus. We use the fact that any parametrized ZX-diagram admits a linear map interpretation and the derivative of a parametrized linear map is well-defined. Therefore, in place of product rules we require the coherence between the derivatives in two categories related by an interpretation functor:

▶ **Definition 14** (Interpretation-coherent derivative)**.** *For two categories $\mathcal{A}, \mathcal{B}$ that are related by a standard interpretation $[\![-]\!] : \mathcal{A} \to \mathcal{B}$ a derivative $\partial_A$ in $\mathcal{A}$ is an unary operator that commutes with the standard interpretation:*

$$\forall D \in \mathcal{A} : \qquad [\![\partial_A D]\!] = \partial_B [\![D]\!] \tag{9}$$

*where the category $\mathcal{B}$ is equipped with sum monoid and $\partial_B$ is derivative in $\mathcal{B}$ satisfying the Definition 13.*

In the context of ZX-diagrams, the Definition 14 requires the derivative of a parametrized diagram to map to the derivative of the corresponding matrix or, in other terms, to satisfy the property of *diagrammatic differentiation* [23].

## 4.1 Linear diagrams

Between parametrized diagrams, we distinguish the family of linear diagrams:

▶ **Definition 15** (Linear diagrams [14])**.** *A ZX-diagram is linear in $\beta_1, \ldots, \beta_k$ with constants in $L \subset \mathbb{R}$ if it is generated by $\alpha$, $\alpha$, , , , , , combined by tensor product and composition with $\alpha$ of the form $\sum_i n_i\beta_i + c$ with $n_i \in \mathbb{Z}$ and $c \in L$.*

It was shown in [14] that for $L = \{\frac{n\pi}{4}\}_{n\in\mathbb{Z}}$ the Clifford+T axiomatization (Figure 1) is complete for linear diagrams.

The family of linear diagrams may appear restricted compared to ZX-diagrams that allow angles from a more general class of functions. It is, however, sufficient for applications in variational quantum algorithms as they use circuits where parameters appear in a linear fashion [3]. More importantly, for this family we demonstrate simple formulas for the derivative. We believe that such formulas are not obtainable even for a slightly more general fragment $\mathrm{ZX}_{\mathcal{A}_n}$ where angles are in the group of affine functions $\mathcal{A}_n = \{(\beta) \to c^T\beta + c_0 | c \in \mathbb{R}^n, c_0 \in \mathbb{R}\}$. Intuitively, the difficulty comes from the absence of a simple representation for a general matrix over real numbers in terms of spiders. This restriction is removed in algebraic ZX-calculus [26] at the cost of an extended set of generators.

In order to keep the notations simple in what follows we restrict our attention to one-variable diagrams $\mathrm{ZX}(\beta)$. We denote the corresponding matrices by $\mathcal{M}(\beta)$. The derivative $\partial_M : \mathcal{M}(\beta) \to \mathcal{M}(\beta)$ is defined by entrywise application of the derivative $\partial_\beta : k\beta + c \mapsto k$. All results may be easily extended to the case of partial derivatives $\partial_{\beta_i}$ for linear diagrams with an arbitrary number of variables.

## 4.2 Diagrammatic differentiation with controlizers

The derivative in $\mathcal{M}(\beta)$ is defined through product rules that involve sums. In this section we use constructions from Section 3 to incorporate these rules in the diagrammatic framework.

In what follows we denote by $C : \mathrm{ZX} \to \mathrm{ZX}$ any map that satisfies Definition 6 of controlizer.

▶ **Definition 16.** *We call $C$-derivative a map $\Delta : \mathrm{ZX}(\beta) \to \mathrm{ZX}(\beta)$ that associates to a diagram $D : n \to m$ another diagram $\Delta(D) : 1 \to n + m$ defined as follows:*

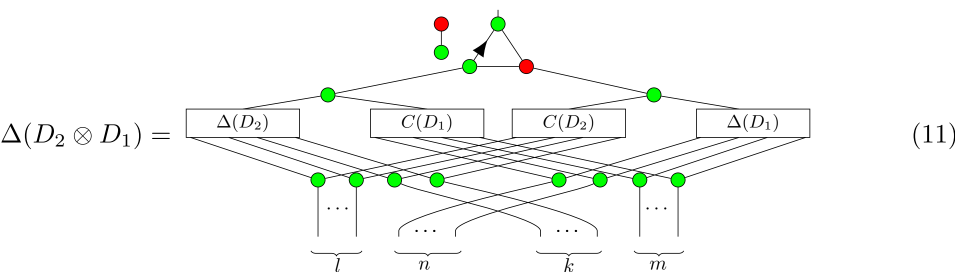
**(i)** ***Generators:*** *For parametrized spiders:* $\Delta\left[\beta\right] = \ldots, \quad \Delta\left[-\beta\right] = \ldots \; \pi - \beta,$

$$\Delta\left[k\beta\right] = \Delta\left[k\{\beta, \ldots, \beta\}\right], \quad \Delta\left[k\beta\right] = \Delta\left[k\beta\right] \tag{10}$$

*For all generators $g : n \to m$ that are independent on $\beta$ $\Delta[g] = \ldots \underbrace{\ldots}_{n+m}$.*

**(ii)** ***Tensor product:*** *for $D_1 : n \to m$ and $D_2 : l \to k$ the diagram $\Delta(D_2 \otimes D_1)$ is:*

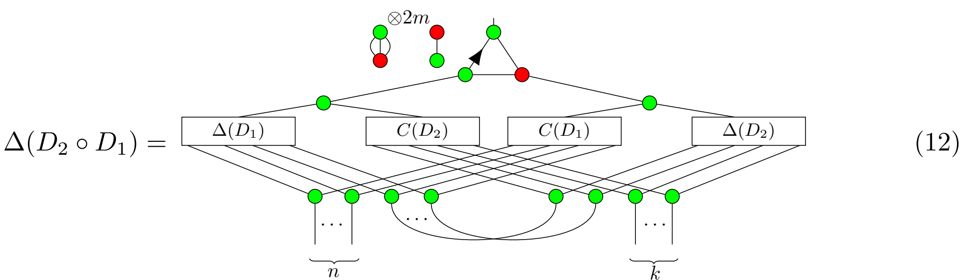
$\Delta(D_2 \otimes D_1) =$




(11)

**(iii)** ***Composition:*** *for $D_1 : n \to m$ and $D_2 : m \to k$ the diagram $\Delta(D_2 \circ D_1)$ is:*

$\Delta(D_2 \circ D_1) =$




(12)

▶ Remark 17. It follows from Lemma 5 that for every diagram $D : n \to m$, $\Delta(D) : 1 \to n+m$ is a controlled state.

The Remark 9 on the dependency of the output on the decomposition order is also true for the map $\Delta$.

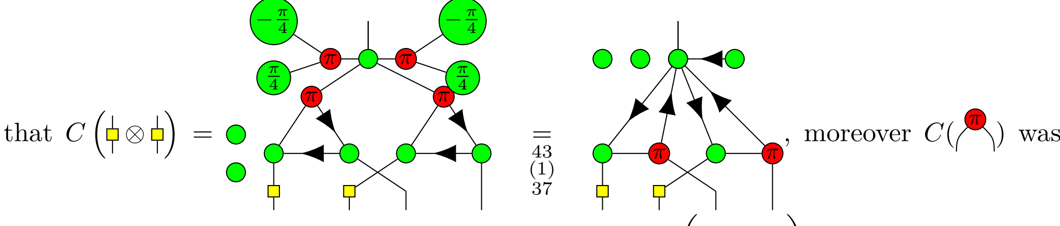
▶ **Definition 18.** *Given the $C$-derivative $\Delta$, let $\partial_C : \mathrm{ZX}(\beta) \to \mathrm{ZX}(\beta)$ be the unary operator such that for any diagram $D : n \to m$,*

$$\partial_C[D] = \quad (13)$$



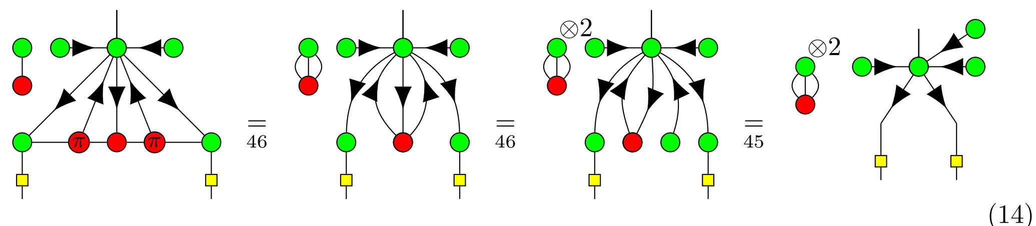
▶ **Theorem 19.** *The operator $\partial_C$ satisfies the Definition 14 of diagrammatic differentiation.*

▶ **Example 20.** We apply Definition 18 to the simple diagram $=$ . Notice that





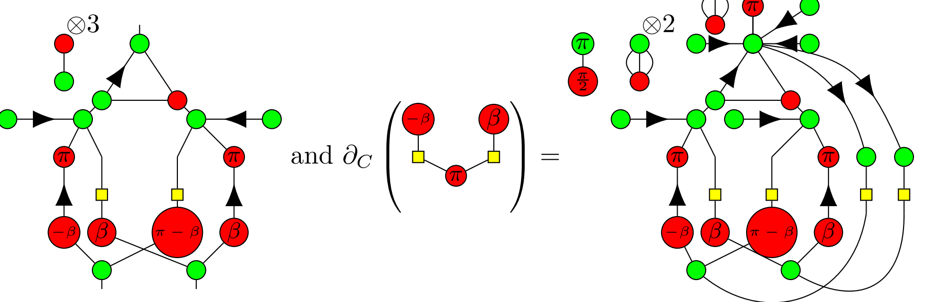
already found in Example 10. We obtain a diagram for $C$ ( π ):





(14)

We know from Lemma 48 that $\Delta$ ( π ) $=$ . By definition, $\Delta$ ( $-\beta$ $\beta$ ) $=$





The last diagram may be further simplified. However, we show later (Example 30) that with our second approach a much simpler diagram for this expression may be obtained directly.

## 4.3 Formula for derivatives in $\mathrm{ZX}(\beta)$

Although perfectly correct, the differentiation procedure described above leads to very puzzling output even for small diagrams (see Example 20). In this section we provide a simpler approach to obtain the derivative of a diagram in $\mathrm{ZX}(\beta)$. We formalize it in definitions $\partial_{\mathrm{ZX}}$ and $\partial_P$ of unary operators that satisfy the property of diagrammatic differentiation (Definition 14).

Let's denote by $X_\beta(n,m)$ diagrams $\underbrace{\beta \ldots \beta}_{n}\ \underbrace{-\beta \ldots -\beta}_{m}$ from $\mathrm{ZX}(\beta)$.

From the (S1) and (H) rules and the paradigm *Only topology matters* follows:

▷ Claim 21. Using the rules of ZX calculus, each diagram $D(\beta): i \to o$ from $ZX(\beta)$ may be transformed into the form

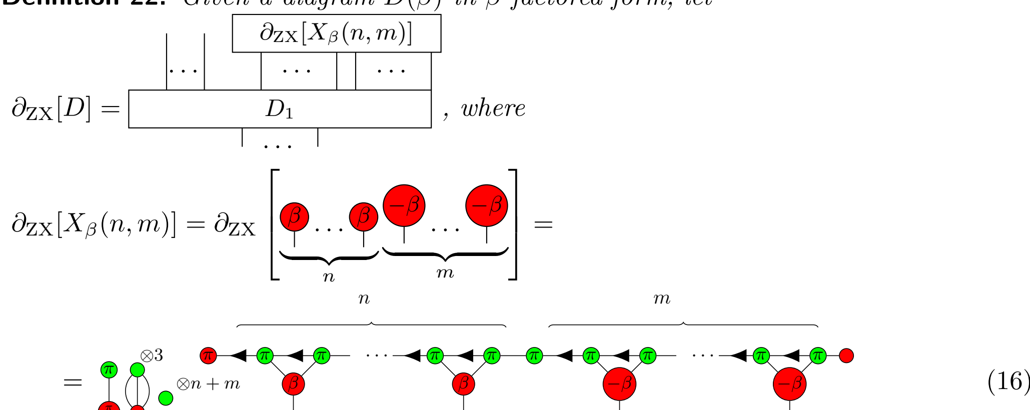
$$D(\beta) = \quad \overbrace{\ldots}^{i}\ \overbrace{\beta \ldots \beta}^{n}\ \overbrace{-\beta \ldots -\beta}^{m} \text{ into } D_1 \text{ with outputs } \underbrace{\ldots}_{o} \tag{15}$$

where $n$, $m$ are some integer numbers and $D_1 : i + n + m \to o$ is constant with respect to $\beta$. We call diagrams in this form $\beta$-factored.

A rigorous demonstration of the claim 21 may be found in [14].

We define the derivative for diagrams in $\beta$-factored forms:

▶ **Definition 22.** *Given a diagram $D(\beta)$ in $\beta$-factored form, let*




▶ **Theorem 23.** *The operator $\partial_{\mathrm{ZX}}[-]$ from the Definition 22 satisfies the property of diagrammatic differentiation:*

*For any diagram $D(\beta) \in \mathrm{ZX}(\beta)$ in $\beta$-factored form $[\![\partial_{\mathrm{ZX}} D(\beta)]\!] = \partial_M [\![D(\beta)]\!]$*

We remark that according to the Definition 14 the derivative $D' : n \to m$ of a diagram $D : n \to m$ that is constant on $\beta$ is such that $[\![D']\!] = \partial_M [\![D]\!] = (0)_{n \times m}$. Therefore, Theorem 23 is a direct consequence of the following lemma:

▶ **Lemma 24.** *For any $n, m$:*

$$[\![\partial_{\mathrm{ZX}} X_\beta(n,m)]\!] = \partial_M [\![X_\beta(n,m)]\!] \tag{17}$$

**Proof (Lemma 24).** We prove the lemma by induction. The demonstration is done for the induction over $n$, the proof for $m$ is directly obtainable in the same way.

**Base.** We show that $[\![\partial_{\mathrm{ZX}} X_\beta(1,0)]\!] = \partial_M [\![\,\beta\,]\!] = \partial_M\left(|+\rangle + e^{i\beta}|-\rangle\right) = ie^{i\beta}|-\rangle$. Indeed,

$$[\![\partial_{\mathrm{ZX}} X_\beta(1,0)]\!] = [\![\,\cdots\,]\!] \underset{35}{\overset{47}{=}} [\![\,\cdots\,]\!] = ie^{i\beta}|-\rangle$$

**Step.** By induction, we assume that the equation (17) holds for some $n$ and $m$. We show that, under this assumption, $[\![\partial_{\mathrm{ZX}} X_\beta(n+1,m)]\!] = \partial_M [\![X_\beta(n+1,m)]\!]$. The demonstration follows from the claims below:

▷ Claim 25.

$$\partial_M [\![X_\beta(n+1,m)]\!] = [\![\partial_{\mathrm{ZX}} [X_\beta(1,0)] \otimes X_\beta(n,m)]\!] + [\![X_\beta(1,0) \otimes \partial_{\mathrm{ZX}} [X_\beta(n,m)]]\!] \tag{18}$$

where $(+)$ is the sum in $\mathcal{M}(\beta)$.

▷ Claim 26. We can find a controlled state $\tilde{X} : 1 \to n+1+m$ and a constant scalar $c \in \mathrm{ZX}_{\frac{\pi}{2}}$ such that

$$\left[\!\!\left[ c \otimes \left( \tilde{X} \circ \left[\,\cdots\,\right] \right) \right]\!\!\right] = [\![\partial_{\mathrm{ZX}} [X_\beta(1,0)] \otimes X_\beta(n,m)]\!] + [\![X_\beta(1,0) \otimes \partial_{\mathrm{ZX}} [X_\beta(n,m)]]\!] \tag{19}$$

▷ Claim 27.

$$\mathrm{ZX} \vdash \partial_{\mathrm{ZX}} X_\beta(n+1,m) = c \otimes \left( \tilde{X} \circ \left[\,\cdots\,\right] \right) \tag{20}$$

◀

## 4.4 Simplified formula for paired spiders

Variational quantum algorithms use gradients in the search for optimal parameter values. The objective minimized by these algorithms can be expressed as $\langle\psi(\beta)|H|\psi(\beta)\rangle$ where the diagram for $\langle\psi(\beta)| = (|\psi(\beta)\rangle)^\dagger$ is obtained out of the diagram for $|\psi(\beta)\rangle$ by flipping up side down followed by the change of signs in spiders. Therefore, parameters in the diagram for $\langle\psi(\beta)|H|\psi(\beta)\rangle$ appear in pairs $-\beta$ $\beta$.

We suggest a more compact formula for diagrams in what we call *pair-factored form*: $D_2 \circ (D_1 \otimes Y(n))$. In this expression $Y_\beta(n) = \underbrace{\left(-\beta \quad \beta\right) \ldots \left(-\beta \quad \beta\right)}_{n}$.

▶ **Lemma 28.** *The diagram:*

$$\partial_P(Y_\beta(n)) = \cdots \tag{21}$$

*satisfies* $[\![\partial_P(Y_\beta(n))]\!] = \partial_M [\![Y_\beta(n)]\!]$.

We prove Lemma 28 by applying the same approach as in the proof of Lemma 24. We can then replace by (21) the expression (16) in Definition 22 and obtain the derivative for diagrams in pair-factored form.

▶ **Observation 29.** *It is possible to extend Lemma 28 to find the derivative for $X_\beta(n,m)$ when $n \neq m$. Indeed, using the fact that* [diagram] = [empty diagram] *we can balance the number of $\beta$ and $-\beta$. For instance, if $n > m$: $\partial_P\left(X_\beta(n,m)\right) = \sigma \circ$* [diagram: $\otimes n-m$, $\partial_P(Y_\beta(n))$, $n-m$, $2m$] *where $\sigma$ is some wire permutation and*

$$[\text{diagram}] \underset{35}{\overset{(B1)}{=}} [\text{diagram}] \tag{22}$$

▶ **Example 30.** We apply Lemma 28 to the same diagram as in Example 20:

$$\partial_P\left([\text{diagram}]\right) = [\text{diagram}] = [\text{diagram}] = [\text{diagram}] \tag{23}$$

## 5 Diagrammatic representation of Ising Hamiltonians

Parametrized quantum circuits are the main component of quantum-classical variational algorithms such as QAOA [7] and VQE [20]. These algorithms are designed to (approximately) solve problems of optimization over binary variables:

$$\min_{x\in\{0,1\}^n} f(x) \tag{24}$$

In order to be treated by a quantum computer an instance $f : \{0,1\}^n \to \mathbb{R}$ of the optimization problem (24) is encoded in a Hamiltonian - an operator $H_f$ acting on qubit states. The Hamiltonian is diagonal in computational basis, $H_f : |x\rangle \to f(x)|x\rangle$. The ground state of $H_f$ corresponds to the optimum of the problem.

For every input Hamiltonian $H_f$ a quantum-classical optimization algorithm starts by designing an anzatz $Q_f(\beta) : n \to n$ [3]. An anzatz is a parametrized quantum circuit with blocks that (possibly) depend on $H_f$. Classical optimization is used to determine the values $\hat{\beta}$ that minimize the expectation of the Hamiltonian $\langle\psi(\hat{\beta})|H_f|\psi(\hat{\beta})\rangle$ [7].

Many important optimization problems such as *Maximum Cut* and *Maximum Independent Set* in a graph may be encoded in so called Ising Hamiltonians [18]:

▶ **Definition 31.** *An Ising Hamiltonian $H : n \to n$ with integer coefficients is an operator:*

$$H = \sum_{1\le i\le n} h_i Z_i + \sum_{1\le i<j\le n} h_{ij} Z_i Z_j, \quad h_i, h_{ij} \in \mathbb{Z} \tag{25}$$

*where $Z_i$ denotes Pauli-Z gate acting on the qubit $i$.*

We observe that there is no direct way to transform the definition of the Hamiltonian (31) to a ZX-diagram. Indeed, Hamiltonian is a non-unitary matrix equal to a *sum* of Pauli gates that is inherently difficult to represent as a diagram. So far, all attempts in this direction used formal sums of diagrams [23, 28]. As a consequence, the application of ZX-calculus to variational algorithms was limited. We show how our formula (16) allows to find a diagram for an Ising Hamiltonian $H$.

Firstly, we remark that for an Ising Hamiltonians $H$ the diagram $D_U(\beta)$ of the linear map $U(\beta) = e^{i\beta H}$ is easy to find [24]. For Hamiltonians with integer coefficients the matrix $U(\beta) = e^{i\beta H}$ belongs to $\mathcal{M}(\beta)$. It satisfies the definition of strongly continuous one-parameter unitary group:

▶ **Definition 32** (Unitary group [23])**.** *A one-parameter unitary group is a unitary matrix* $U : n \to n$ *in* $\mathcal{M}(\beta)$ *with* $U(0) = id_n$ *and* $U(\beta)U(\beta') = U(\beta + \beta')$ *for all* $\beta, \beta' \in \mathbb{R}$*. It is strongly continuous when* $\lim_{\beta \to \beta_0} U(\beta) = U(\beta_0)$ *for all* $\beta_0 \in \mathbb{R}$*.*

▶ **Theorem 33** (Stone ([22]))**.** *There is a one-to-one correspondence between strongly continuous one-parameter unitary groups* $U : n \to n$ *in* $\mathcal{M}(\beta)$ *and self-adjoint matrices* $H : n \to n$ *in* $\mathcal{M}$*. The bijection is given explicitly by* $U(\beta) = e^{i\beta H}$ *and* $H = -i(\partial_M U)(0)$*.*

We use the bijection from the Stone's theorem to find the diagram $h \in ZX_{\mathbb{R}}$ such that $[\![h]\!] = H$. Using the property $U(0) = \mathrm{id}_n$ we obtain:

$$H = -i[\partial_M U(\beta)](0) = -i \otimes [\![ [\partial_{\mathrm{ZX}} D_U](\beta) ]\!](0) = -i \otimes [\![ [\partial_{\mathrm{ZX}} D_U](0) ]\!]$$
$$= [\![ \cdots [\partial_{\mathrm{ZX}} D_U](0) ]\!] = [\![h]\!] \tag{26}$$

where the third equality is due to the fact that the evaluation commutes with the standard interpretation.

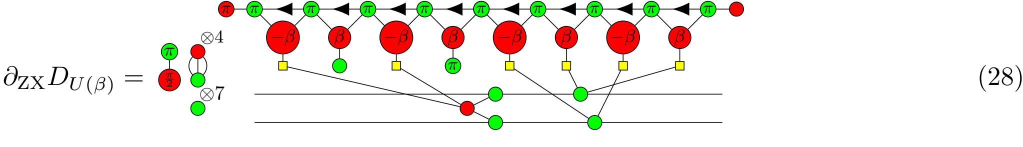
We give an example of diagram for an Ising Hamiltonian obtained via our approach.

▶ **Example 34.** Let $H : 2 \to 2$, $H = Z_1 - Z_2 + Z_1 Z_2$. The diagram $D_U(\beta)$ for $U(\beta) = e^{i\beta H}$ is:

$$D_U(\beta) = \cdots \tag{27}$$

Using the formula (21) we find the derivative of $D_{U(\beta)}$:

$$\partial_{\mathrm{ZX}} D_{U(\beta)} = \tag{28}$$



$$h = \cdots [\partial_{\mathrm{ZX}} D_{U(\beta)}]_{\beta \to 0} = \cdots \tag{29}$$

## 6 Discussions

In this work, we have introduced for the first time an inductive definition for addition of ZX-diagrams, that we have then used to introduce an inductive definition of the differentiation of ZX-diagrams. Addition and differentiation are essential tools for the development and the study of quantum algorithms, but, as a matter of fact, both of them are leading to large diagrams, even when the initial diagrams are fairly simple. From a process theory point of

view, contrary to sequential and parallel compositions, the addition is not physical operation, hence it is not surprising that it is not a native or simple operation over ZX-diagrams. The good news is that we can rely on the powerful equation theory of the ZX-calculus to simplify, when it is possible, the diagrams representing the sum or the differentiation of diagrams.

In Section 4.3, we have shown that instead of simplifying the resulting diagrams *a posteriori*, one can *a priori* put the initial diagrams in an appropriate form. While this approach is not inductive anymore, it seems to ease the differentiation of diagrams in practice. Notice that this last approach, in particular Definition 22, leads to a very similar differentiation of diagrams to the one independently introduced in [27]. In their work, the authors directly introduce the differentiation of ZX-diagrams in some particular form, in contrast to the inductive definition we propose, but another important difference is actually the diagrammatic language and its expressivity. While our work is based on the "vanilla" ZX-calculus, the authors of [27] rely on the algebraic ZX-calculus, i.e. a ZX-calculus augmented with boxes allowing, roughly speaking, the direct representation of a complex numbers, whereas only angles can be used as parameters in the vanilla ZX-calculus. As a consequence when an algebraic ZX-diagram is parameterised by an arbitrary derivable function $f(x)$, the differentiated algebraic ZX-diagram is parametrised by $f'(x)$. Such an approach is not possible in the more constrained vanilla ZX-calculus thus we restrict our attention to a family of functions (essentially the linear ones) which derivative can be expressed using the structure of the vanilla ZX-calculus.

In most practical examples the vanilla ZX-calculus is sufficient to represent parametrised computation. As an application we have shown that our result allows the construction of diagrams for Ising Hamiltonians and for derivatives of parametrized circuits. Therefore, it becomes possible to study variational algorithms entirely within the ZX-calculus. In particular, we can use rewrite rules to simplify such expressions as $\langle\psi(\hat{\beta})|H_f|\psi(\hat{\beta})\rangle$ and $\frac{\partial\langle\psi(\hat{\beta})|H_f|\psi(\hat{\beta})\rangle}{\partial\beta}$. We believe that it will lead to a better understanding of the potential of variational algorithms and of their applications to real-world problems.

## References


1. Miriam Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, September 2014. doi:10.1088/1367-2630/16/9/093021.
2. R. F. BLUTE, J. R. B. COCKETT, and R. A. G. SEELY. Differential categories. *Mathematical Structures in Computer Science*, 16(6):1049–1083, 2006. doi:10.1017/S0960129506005676.
3. M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, August 2021. doi:10.1038/s42254-021-00348-9.
4. Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
5. Bob Coecke and Aleks Kissinger. *Picturing Phases and Complementarity*, pages 510–623. Cambridge University Press, 2017. doi:10.1017/9781316219317.010.
6. Ross Duncan and Simon Perdrix. Pivoting makes the ZX-calculus complete for real stabilizers. *Electronic Proceedings in Theoretical Computer Science*, 171:50–62, December 2014. doi:10.4204/eptcs.171.5.
7. Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014. arXiv:1411.4028.

**8** Gian Giacomo Guerreschi and Mikhail Smelyanskiy. Practical optimization for hybrid quantum-classical algorithms. *arXiv: Quantum Physics*, 2017.

**9** Stuart Hadfield. On the Representation of Boolean and Real Functions as Hamiltonians for Quantum Computing. *ACM Transactions on Quantum Computing*, 2(4):1–21, December 2021. `doi:10.1145/3478519`.

**10** Amar Hadzihasanovic. A diagrammatic axiomatisation for qubit entanglement. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 573–584, 2015. `doi:10.1109/LICS.2015.59`.

**11** Amar Hadzihasanovic, Kang Feng Ng, and Quanlong Wang. Two complete axiomatisations of pure-state qubit quantum computing. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 502–511, New York, NY, USA, 2018. ACM. `doi:10.1145/3209108.3209128`.

**12** Emmanuel Jeandel, Simon Perdrix, and Margarita Veshchezerova. Addition and Differentiation of ZX-diagrams, 2022. `doi:10.48550/ARXIV.2202.11386`.

**13** Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. A Complete Axiomatisation of the ZX-Calculus for Clifford+T Quantum Mechanics. In *The 33rd Annual {ACM/IEEE} Symposium on Logic in Computer Science, {LICS} 2018*, Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pages 559–568, Oxford, United Kingdom, July 2018. `doi:10.1145/3209108.3209131`.

**14** Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Diagrammatic Reasoning beyond Clifford+T Quantum Mechanics. In *The 33rd Annual Symposium on Logic in Computer Science*, Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pages 569–578, Oxford, United Kingdom, July 2018. `doi:10.1145/3209108.3209139`.

**15** Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. A Generic Normal Form for ZX-Diagrams and Application to the Rational Angle Completeness. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '19. IEEE Press, 2019.

**16** Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of the ZX-Calculus. *Logical Methods in Computer Science*, Volume 16, Issue 2, June 2020. `doi:10.23638/LMCS-16(2:11)2020`.

**17** Emmanuel Jeandel, Simon Perdrix, Renaud Vilmart, and Quanlong Wang. ZX-Calculus: Cyclotomic Supplementarity and Incompleteness for Clifford+T Quantum Mechanics. In Kim G. Larsen, Hans L. Bodlaender, and Jean-Francois Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.MFCS.2017.11`.

**18** Andrew Lucas. Ising formulations of many np problems. *Frontiers in Physics*, 2, 2014. `doi:10.3389/fphy.2014.00005`.

**19** Kang Feng Ng and Quanlong Wang. Completeness of the ZX-calculus for Pure Qubit Clifford+T Quantum Mechanics, 2018. `arXiv:1801.07993`.

**20** Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):4213, July 2014. `doi:10.1038/ncomms5213`.

**21** John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. `doi:10.22331/q-2018-08-06-79`.

**22** M. H. Stone. On One-Parameter Unitary Groups in Hilbert Space. *Annals of Mathematics*, 33(3):643–648, 1932. URL: `http://www.jstor.org/stable/1968538`.

**23** Alexis Toumi, Richie Yeung, and Giovanni de Felice. Diagrammatic Differentiation for Quantum Machine Learning. *arXiv e-prints*, page arXiv:2103.07960, March 2021. `arXiv:2103.07960`.

**24** John van de Wetering. ZX-calculus for the working quantum computer scientist, 2020. `arXiv:2012.13966`.

25 Renaud Vilmart. A near-optimal axiomatisation of ZX-calculus for pure qubit quantum mechanics. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019. `arXiv:arXiv:1812.09114`.

26 Quanlong Wang. An Algebraic Axiomatisation of ZX-calculus. *Electronic Proceedings in Theoretical Computer Science*, 340:303–332, September 2021. `doi:10.4204/eptcs.340.16`.

27 Quanlong Wang and Richie Yeung. Differentiating and Integrating ZX Diagrams, 2022. `arXiv:2201.13250`.

28 Richie Yeung. Diagrammatic design and study of ansätze for quantum machine learning, 2020. `arXiv:2011.11073`.

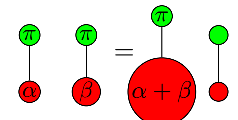
29 Chen Zhao and Xiao-Shan Gao. Analyzing the barren plateau phenomenon in training quantum neural networks with the ZX-calculus. *Quantum*, 5:466, June 2021. `doi:10.22331/q-2021-06-04-466`.


# A ZX lemmas

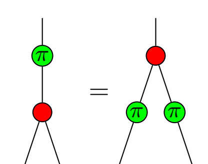
## A.1 Already proven lemmas [15]

▶ **Lemma 35.**

▶ **Lemma 36.**




▶ **Lemma 37.**

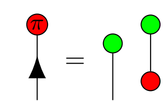



▶ **Lemma 38.**

▶ **Lemma 39.**

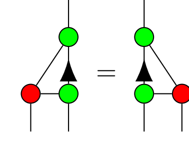
▶ **Lemma 40.**

▶ **Lemma 41.**

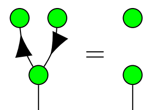



▶ **Lemma 42.**

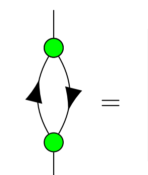
▶ **Lemma 43.**



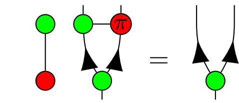

▶ **Lemma 44.**



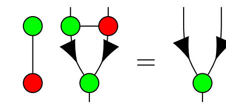

▶ **Lemma 45.**




▶ **Lemma 46.**

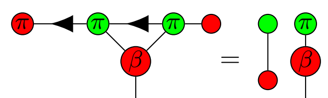



*and*

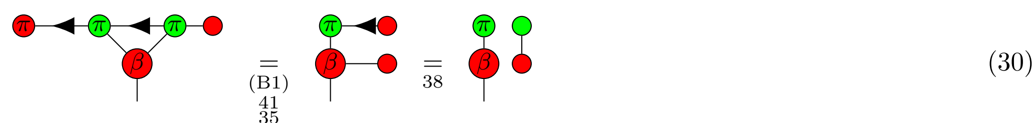



## A.2 New lemmas

▶ **Lemma 47.**



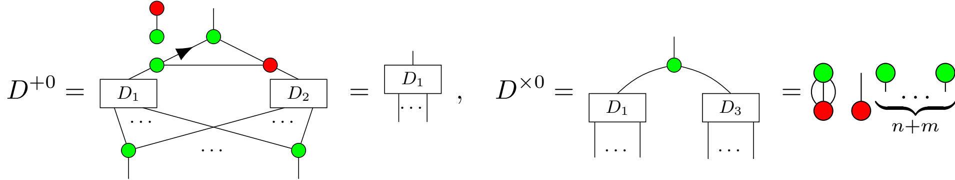
**Proof.**



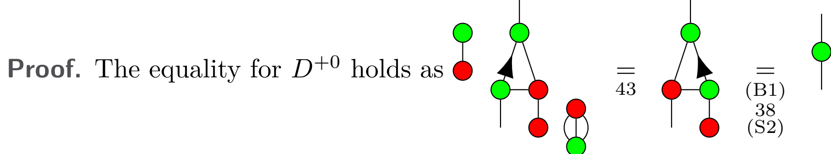

(30)

◀

▶ **Lemma 48.** *For all controlled states* $D : 1 \to n$ *and states* $D_2$, $D_3$ *that are* $D_2 =$ $\underbrace{\dots}_{n}$ *and* $D_3 =$ $\underbrace{\dots}_{m}$*:*




**Proof.** The equality for $D^{+0}$ holds as




The equality for $D^{\times 0}$ follows from (B1) and the definition (3) of controlled states. ◀

# Restricting Tree Grammars with Term Rewriting

**Jan Bessai** ✉
TU Dortmund, Germany

**Lukasz Czajka** ✉
TU Dortmund, Germany

**Felix Laarmann** ✉
TU Dortmund, Germany

**Jakob Rehof** ✉
TU Dortmund, Germany

## Abstract

We investigate the problem of enumerating all terms generated by a tree-grammar which are also in normal form with respect to a set of directed equations (rewriting relation). To this end we show that deciding emptiness and finiteness of the resulting set is EXPTIME-complete. The emptiness result is inspired by a prior result by Comon and Jacquemard on ground reducibility. The finiteness result is based on modification of pumping arguments used by Comon and Jacquemard. We highlight practical applications and limitations. We provide and evaluate a prototype implementation. Limitations are somewhat surprising in that, while deciding emptiness and finiteness is EXPTIME-complete for linear and nonlinear rewrite relations, the linear case is practically feasible while the nonlinear case is infeasible, even for a trivially small example. The algorithms provided for the linear case also improve on prior practical results by Kallat et al.

**2012 ACM Subject Classification** Theory of computation → Tree languages; Theory of computation → Automata extensions; Theory of computation → Equational logic and rewriting

**Keywords and phrases** tree automata, tree grammar, term rewriting, normalization, emptiness, finiteness

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.14

**Acknowledgements** We thank Christoph Stahl for creating the `tikz` figures. We also thank our reviewers for their insightful and useful comments which improved the final version of the paper.

## 1 Introduction

Suppose we are given a tree grammar $G$ over a ranked alphabet $\mathcal{F}$ and a rewriting relation $R$ over terms generated from $\mathcal{F}$. We are interested in deciding emptiness and finiteness of the set $L(G) \cap \mathrm{NF}(R)$, where $\mathrm{NF}(R)$ is the set of terms in normal form with respect to $R$. This problem may arise naturally in situations where trees recognized by $G$ are subject to simplifications under $R$ and we are only interested in simplified terms. For example, we may think of $G$ as recognizing a language of algebraic expressions including, say, expressions of the form $f(a, b)$, and $R$ captures simplifications under algebraic laws, say, idempotence $f(X, X) \to X$.

Our interest in this problem arose in the context of work on component-based synthesis [18], specifically combinatory logic synthesis (CLS). CLS is based on solving bounded versions of the inhabitation problem for combinatory logic with intersection types [17, 8] and has been implemented in the CLS-framework (see [3] for a fairly recent description). CLS has been applied in a number of contexts, recent examples include [4, 21, 10, 19].

In CLS, the (possibly infinite) solution set to a synthesis query is a set of combinatory terms (each representing a program or a metaprogram), which is represented by a tree grammar $G$ recognizing combinatory terms. Here, $R$ acts as a filter restricting the solution

© Jan Bessai, Lukasz Czajka, Felix Laarmann, and Jakob Rehof;
licensed under Creative Commons License CC-BY 4.0

set to normal forms in $L(G) \cap \mathrm{NF}(R)$, and we are interested in enumerating normal solutions. Since the filter specified by $R$ might well lead to a finite set of normal solutions even though $L(G)$ is infinite, knowing whether $L(G) \cap \mathrm{NF}(R)$ is empty or finite is of immediate interest. The results reported in the present paper form the basis of a prototype implementation intended to become an extension to the CLS-framework.

Notice that the problems considered here are entirely different from the problem of recognizing the normal forms (wrt. $R$) of $L(G)$ for a given grammar of terms $G$. The latter problem is obviously undecidable (take $G$ to recognize a given $SKI$-term, and we would need to solve the halting problem for $SKI$-calculus), but it is also not relevant for our purposes, since we are interested only in terms that are already contained in the solution set $L(G)$. In our setting, the rewriting relation $R$ is used as a filter such that only the left-hand sides of rules matter to filter out non-normal forms from $L(G)$ (essentially, by solving the problem of non-matchability of terms with any left-hand side of $R$).

## 1.1 Contributions

Our contribution is twofold. First, we prove EXPTIME-completeness of emptiness and finiteness of $L(G) \cap \mathrm{NF}(R)$. Our techniques draw on previous work by Comon and Jacquemard (see Section 1.2) on automata with disequality constraints (ADC) for the ground reducibility problem. Disequality constraints are necessary to handle nonlinear rules in $R$. Our main technical contribution is contained in the Bound Theorem (Theorem 7), which provides a bound on the maximum height of accepted terms, when $L(G) \cap \mathrm{NF}(R)$ is finite. The bound follows from a pumping argument for finiteness and acts as an upper bound for enumeration in the finite case.

Second, we provide experimental analysis of the algorithm for deciding emptiness and finiteness provided here, based on a Haskell implementation. It turns out that, even though the left-linear restriction (wrt. $R$) is somewhat surprisingly already EXPTIME-complete for both problems (Theorem 23, Theorem 25), the performance in the nonlinear case is orders of magnitude worse than in the linear case. Our analysis shows that the nonlinear case reaches an order of magnitude of worst-case performance (rendering it infeasible for even trivially small examples), whereas the linear restriction can be engineered to be practically feasible, improving on a previously published algorithm. Whether one can find heuristics to engineer the nonlinear case for practically interesting cases is left as a question for future research.

## 1.2 Related work

The theoretical results in the present paper are adaptations and extensions of the results of Comon and Jacquemard on the EXPTIME-completeness of the ground reducibility problem [7, 6]. We consider different problems of emptiness and finiteness of the intersection of a regular tree language with the set of normal forms of a rewrite system. While the proof of our Bound Theorem and the automata constructions draw heavily from [7], the adaptation of the results to emptiness and finiteness is not trivial.

The EXPTIME-completeness of the emptiness problem was essentially shown by Comon and Jacquemard [7] (only relatively small adjustments are necessary to adapt their arguments to our problem). An EXPTIME algorithm for finiteness was essentially already obtained in [9], where [9, Lemma 5.19] corresponds to our Bound Theorem and the constructions in the proofs are similar. However, our exponential bound is better than the exponential bound given in [9], which may have practical implications. EXPTIME-hardness of the finiteness problem seems to be new.

Our results depend on the notion of automata with disequality constraints (ADCs) introduced by Comon and Jacquemard [7]. Related automata frameworks are tree automata with normalization [16] and equational tree automata [15]. In these frameworks, the automata transitions are defined modulo normalization or an equational theory resulting in accepted languages closed under these operations, while we are interested in restrictions of regular tree languages to normal forms of a rewrite system. Another related model is tree automata with global constraints [13] where the constraints are associated with pairs of states and enforce equality or disequality of all subterms at all nodes where the states appear in the corresponding run, in contrast to ADCs where the constraints are local and associated with transition rules, enforcing disequality of subterms at a given transition.

Going beyond the previously described usecase for CLS, other synthesis frameworks might profit from our approach. For example, Madhusudan [14] describes a framework for synthesizing reactive programs. This approach is similar to recent additions to the broader field of syntax guided synthesis [12]. In both cases, synthesized programs are represented by trees and constructed from tree-languages, that are then restricted to match desired program semantics. In the present paper we are not concerned with arbitrary semantic specifications, but just equations for program normalization. In synthesis frameworks such as the above, this might be a useful way to reduce the search space or filter solutions.

## 2 Preliminaries

In this section we fix notations and recall standard definitions related to tree grammars and term rewriting. See e.g. [5] and, respectively, [1, 20] for more thorough introductions to these topics.

By $\mathcal{T}(\mathcal{F}, X)$ we denote the set of all first-order terms over the signature $\mathcal{F}$ with variables taken from the set $X$. The set of *ground terms* $\mathcal{T}(\mathcal{F}, \emptyset)$ is also denoted by $\mathcal{T}(\mathcal{F})$. By $\epsilon$ we denote the empty string, by $\cdot$ the concatenation operation on strings, and by $[i]$ the string consisting of a single letter $i$. The set of *positions* of a term $t \in \mathcal{T}(\mathcal{F}, X)$ is a set $\mathrm{Pos}(t)$ of strings of positive integers defined by: (1) if $t = x$ then $\mathrm{Pos}(t) = \{\epsilon\}$; (2) if $t = f(t_1, \ldots, t_n)$ then $\mathrm{Pos}(t) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{[i] \cdot p \mid p \in \mathrm{Pos}(t_i)\}$. The *size* of a term $t$ is the cardinality of $\mathrm{Pos}(t)$. The *prefix order* on positions is defined by: $p \preceq q$ iff there is $p'$ with $p \cdot p' = q$. For $p \in \mathrm{Pos}(t)$, the *subterm* of $t$ at position $p$ is denoted by $t|_p$. By $t(p)$ we denote the symbol in $t$ at position $p$. The *replacement* $t[s]_p$ is the term obtained from $s$ by replacing the subterm at position $p$ with $s$. By $\mathrm{Var}(t)$ we denote the set of variables occurring in $t$. A context $C$ is a term in $\mathcal{T}(\mathcal{F}, X \cup \{\Box\})$ such that $\Box$ occurs in $C$ exactly once. By $C[t]$ we denote the term in $\mathcal{T}(\mathcal{F}, X)$ obtained from $C$ by replacing $\Box$ with $t$.

A *term rewriting system* (TRS) $R$ is a set of rules $t \to s$ such that $\mathrm{Var}(s) \subseteq \mathrm{Var}(t)$ and $t$ is not a variable. We denote by $\to_R$ the *reduction relation* associated with the TRS $R$: $t \to_R s$ iff there is a rule $l \to r \in R$, a context $C$ and a substitution $\sigma$ such that $t = C[\sigma l]$ and $s = C[\sigma r]$. A term $t$ is in *normal form* if there is no $t'$ with $t \to_R t'$. The *size* $||R||$ of the TRS $R$ is the sum of the sizes of the left-hand sides of rules in $R$.

For any binary relation $\to$, by $\to^*$ we denote the transitive-reflexive, and by $\to^+$ the transitive closure of $\to$.

A regular tree grammar is a tuple $G = (S, N, \mathcal{F}, R_G)$ such that $S \in N$ is the start symbol, $N$ is a set of nullary nonterminals, $\mathcal{F}$ is a set of terminals, $R_G$ is a set of production rules of the form $A \to \alpha$ where $A \in N$ and $\alpha \in \mathcal{T}(\mathcal{F} \cup N)$. The *derivation relation* associated with $G$ is defined by: $t \to_G s$ iff there is a rule $A \to \alpha \in R$ and a context $C$ such that $t = C[A]$ and $s = C[\alpha]$. The *language generated* by $G$ is defined by $L(G) = \{t \in \mathcal{T}(\mathcal{F}) \mid S \to_G^+ t\}$.

A finite tree automaton over signature $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where $Q$ is a set of states, $Q_f \subseteq Q$ is a set of final states, and $\Delta$ is a set of transition rules of the form $f(q_1, \dots, q_n) \to q$ with $f \in \mathcal{F}_n$ (i.e. $f$ is an $n$-ary symbol in $\mathcal{F}$), $q, q_1, \dots, q_n \in Q$. The move relation $\to_{\mathcal{A}}$ is defined by: $t \to_{\mathcal{A}} t'$ iff there are a transition rule $l \to r$ and a context $C$ with $t = C[l]$ and $t' = C[r]$. A ground term $t \in \mathcal{T}(\mathcal{F})$ is *accepted* by $\mathcal{A}$ if there is $q_f \in Q_f$ with $t \to^*_{\mathcal{A}} q_f$. The language $L(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all terms accepted by $\mathcal{A}$.

In terms of the recognized languages, finite tree automata and regular tree grammars are equivalent. A *regular tree language* is a language recognized by a finite tree automaton, or equivalently a language generated by a regular tree grammar.

## 3 Automata with disequality constraints

Automata with disequality constraints (ADC) were introduced by Comon and Jacquemard in [6, 7]. These are essentially tree automata where some rules may additionally check whether two subterms at given positions are not equal. The idea is to construct a normal forms ADC which recognises exactly the normal forms of a given term rewriting system. The disequality constraints are needed to handle non-left-linear rules. To check emptiness or finiteness of the intersection, a product automaton is created. The construction of the normal forms ADC has already been presented by Comon and Jacquemard. In this section, we recall the definition of ADCs and related notions. The constructions of the normal forms automaton and the finiteness checking algorithms are presented in subsequent sections.

Definitions in this section are either verbatim copies or minor modifications of those in [7].

▶ **Definition 1.** *An* automaton with disequality constraints *(ADC) is a tuple* $(Q, Q^f, \Delta)$ *where $Q$ is a finite set of states, $Q^f \subseteq Q$ is the set of final states, and $\Delta$ is a finite set of transition rules of the form $f(q_1, \dots, q_n) \xrightarrow{c} q$ where $f \in \mathcal{F}^n$, $q_1, \dots, q_n, q \in Q$ and $c$ is a Boolean combination without negation of constraints $p_1 \neq p_2$ with $p_1, p_2$ positions. A term $t \in \mathcal{T}(\mathcal{F})$* satisfies the constraint *$p_1 \neq p_2$, denoted $t \models p_1 \neq p_2$, if both $p_1, p_2 \in \mathrm{Pos}(t)$ and $t|_{p_1} \neq t|_{p_2}$. A* run *of an automaton $\mathcal{A} = (Q, Q^f, \Delta)$ on a term $t$ is a term $\rho$ over signature $\Delta$ (i.e. each rule $r = (f(q_1, \dots, q_n) \to q) \in \Delta$ is treated as an $n$-ary symbol) such that for all $p \in \mathrm{Pos}(t)$, if $t(p) = f \in \mathcal{F}^n$ then $\rho(p)$ is a rule $f(q_1, \dots, q_n) \xrightarrow{c} q$ and:*

1. *$\rho(p \cdot [i])$ is a rule with target $q_i$, for $i = 1, \dots, n$ (weak),*
2. *$t|_p \models c$ (strong).*

*If only the first condition (weak) is satisfied by $\rho$, then $\rho$ is a* weak run.

*A ground term $t \in \mathcal{T}(\mathcal{F})$ is* accepted *by $\mathcal{A}$ if there is a run $\rho$ of $\mathcal{A}$ on $t$ such that $\rho(\epsilon)$ is a rule whose target is a final state in $Q^f$. The* language *$L(\mathcal{A})$ of $\mathcal{A}$ is the set of terms accepted by $\mathcal{A}$.*

▶ Note 2.

- An ADC with all constraints $\top$ is a finite tree automaton (the constraints are always satisfied).
- An ADC can be non-deterministic (more than one run on some term) or not completely specified (no run on some term).
- The term used in the construction of a run $\rho$ is denoted as the associated term$(\rho) \in \mathcal{T}(\mathcal{F})$.

▶ **Example 3.** Let $\mathcal{F} = \{f, a, b\}$ and $Q = \{q\} = Q^f$.

$$\Delta = \{r_1 : a \to q, r_2 : b \to q, r_3 : f(q, q) \xrightarrow{1 \neq 2} q\}.$$

The term $f(a,b)$ is accepted because $\rho = r_3(r_1, r_2)$ is a run on $t$ and $r_3$ yields a final state. The term $f(a,a)$ is not accepted: there is a weak run $r_3(r_1, r_1)$ but the disequality of $r_3$ is not satisfied. In general, the automaton accepts ground terms irreducible by a TRS with a single rule with the left-hand side $f(x,x)$.

▶ **Definition 4.** *Let $\mathcal{A}$ be an ADC.*

*Let $\mathcal{C}(\mathcal{A})$ be the set of all triples $(\beta, \pi, \pi')$ such that $\beta$ is a prefix of $\pi'$ and $\pi \neq \pi'$ or $\pi' \neq \pi$ is an atom occurring in a constraint of transition rules of $\mathcal{A}$. Let $c(\mathcal{A}) = |\mathcal{C}(\mathcal{A})|$.*

*Let $\mathcal{S}(\mathcal{A})$ be the set of all suffixes of positions $\pi, \pi'$ in an atom $\pi \neq \pi'$ occurring in a constraint of a rule in $\mathcal{A}$. Let $s(\mathcal{A}) = |\mathcal{S}(\mathcal{A})|$.*

*We define $d(\mathcal{A})$ as the maximum length of $\pi$ in a constraint $\pi \neq \pi'$ or $\pi' \neq \pi$ in $\mathcal{A}$. By $n(\mathcal{A})$ we denote the maximum number of atomic constraints occurring in a rule of $\mathcal{A}$.*

Note that $c(\mathcal{A}), s(\mathcal{A}) \leq |\mathcal{A}|^2$ and $d(\mathcal{A}), n(\mathcal{A}) \leq |\mathcal{A}|$ and $d(\mathcal{A}) \leq s(\mathcal{A})$. In [7], $c(\mathcal{A})$ and $\mathcal{C}(\mathcal{A})$ are used instead of $s(\mathcal{A})$ and $\mathcal{S}(\mathcal{A})$. Our definitions of $c(\mathcal{A})$ and $\mathcal{C}(\mathcal{A})$ are modifications of the definitions from [7] to upward pumping.

▶ **Definition 5.** *Let $\mathcal{A} = (Q, Q^f, \Delta)$ be an ADC and $\rho$ a weak run of $\mathcal{A}$ on $t$. An* equality *of $\rho$ is a triple of positions $(p, \pi, \pi')$ such that $p, p \cdot \pi, p \cdot \pi' \in \mathrm{Pos}(t)$, $\pi \neq \pi'$ is in the constraint of $\rho(p)$ and $t|_{p\cdot\pi} = t|_{p\cdot\pi'}$.*

*An equality $(p', \pi, \pi')$ in a weak run $\rho$ is classified according to a particular position $p \in \mathrm{Pos}(t)$:*

- *It is* close *to $p$ if $p' \preceq p \prec p' \cdot \pi$ or $p' \preceq p \prec p' \cdot \pi'$,*
- *It is* far *from $p$ if $p' \cdot \pi \preceq p$ or $p' \cdot \pi' \preceq p$.*

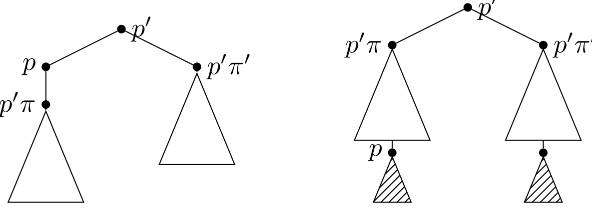



**Figure 1** Equality close to $p$ (left) and equality far from $p$ (right).

▶ **Lemma 6.** *Every equality in $\rho[\rho']_p$ is either far from $p$ or close to $p$.*

**Proof.** Identical to the proof of Lemma 18 in [7]. ◀

## 4 The Bound Theorem

In this section we prove the Bound Theorem which characterises finiteness of the language of an ADC in terms of the maximum height of an accepted term. The theorem is crucial for the correctness of our finiteness checking algorithm.

▶ **Theorem 7** (Bound theorem)**.** *Let $\mathcal{A}$ be an ADC. $L(\mathcal{A})$ is finite iff all accepted terms have have height strictly smaller than*

$$H(\mathcal{A}) = (e+1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A})+1)$$

To prove the theorem, we use pumping arguments similar to that in [7]. Instead of pumping downward decreasing the size of an accepted term, however, we need to pump upward increasing the size arbitrarily. The modifications of the arguments are laborious and not trivial, but they follow closely the proofs in [7]. A similar construction may also be found in [9]. In fact, [9, Lemma 5.19] is a generalisation of our Bound Theorem to a broader class of automata, but with a worse, though still exponential, exact bound.

To fully understand this section, some familiarity with [7] is helpful. We try to convey the underlying intuitions, but we don't see it productive to copy proofs or definitions verbatim where no change is necessary.

In contrast to downward pumping in [7] which uses an arbitrary ordering $\gg$ satisfying the requirements of Section 6, for our upward pumping argument we need the strict embedding ordering $\ggg$ on terms.

▶ **Definition 8.** *An* upward pumping *(wrt. the strict embedding ordering $\ggg$) is a replacement $\rho[\rho']_p$ where $\rho, \rho'$ are runs such that the target state of $\rho'(\epsilon)$ is the same as the target of $\rho(p)$ and $\rho[\rho']_p \ggg \rho$.*

The proofs of the generalised pumping lemmas in [7] are divided into two parts: pumping without creating close equalities and pumping without creating equalities (far or close). The argument for pumping without creating close equalities are adapted to upward pumping, but the complex details need to be checked. The argument for pumping without creating equalities is replaced by a simpler argument for upward pumping, because if we can pump upward without creating close equalities then we can increase the size of the pumping arbitrarily to prevent any far equalities from being created.

▶ **Definition 9.** *Given $\mathcal{A} = (Q, Q^f, \Delta)$ and an integer $k$ we set (where $e$ is Euler's number):*

$$g(\mathcal{A}, k) = (e \times k + 1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})!$$

The following is the main pumping lemma needed in the proof of the Bound Theorem. The proof of this lemma occupies most of this section. It is an analogon of [7, Lemma 19] adapted to upward pumping.

▶ **Lemma 10.** *If $\rho$ is a run of $\mathcal{A}$ and $p_1, \ldots, p_{g(\mathcal{A},k)}$ are positions of $\rho$ such that $\rho|_{p_1} \ggg \ldots \ggg \rho|_{p_{g(\mathcal{A},k)}}$ then there are indices $i_0 < \ldots < i_k$ such that the upward pumping $\rho[\rho|_{p_{i_0}}]_{p_{i_j}}$ does not contain any equality close to $p_{i_j}$.*

▶ **Definition 11.** *Given $p \in \mathrm{Pos}(\rho)$, the set $\mathrm{cr}(p)$ is defined as the set of all triples $(\beta, \pi, \pi')$ such that there is $p' \in \mathrm{Pos}(\rho)$ with $p'\beta = p$ (i.e. $p' = p/\beta$) and $p \prec p'\pi'$ and $\pi \neq \pi'$ or $\pi' \neq \pi$ is a constraint of $\rho(p')$. See Figure 2.*

The intuition is that $\mathrm{cr}(p)$ indicates all possible places above $p$ at which an equality close to $p$ may be created.

▶ **Fact 12.** *If $(p', \pi, \pi')$ is an equality close to $p$, then there is $(\beta, \pi, \pi') \in \mathrm{cr}(p)$ such that $p'\beta = p$.*

▶ **Fact 13.** *For all $p \in \mathrm{Pos}(\rho)$ we have $\mathrm{cr}(p) \subseteq \mathcal{C}(\mathcal{A})$, and thus $|\mathrm{cr}(p)| \leq c(\mathcal{A})$.*

Similarly to [7] we can extract a subsequence $v_0, \ldots, v_{k_2}$ of $p_1, \ldots, p_{g(\mathcal{A},k)}$ such that $\rho(v_0), \ldots, \rho(v_{k_2})$ all have the same target state and $\mathrm{cr}(v_0) = \ldots = \mathrm{cr}(v_{k_2})$, where $k_2 = (e \times k+1) \times c(\mathcal{A})! - 1$. For this purpose, we first extract a subsequence $u_1, \ldots, u_{k_1}$ of $p_1, \ldots, p_{g(\mathcal{A},k)}$ such that all $u_i$ have the same target state, where $k_1 = \frac{g(\mathcal{A},k)}{|Q|} = (e \times k + 1) \times 2^{c(\mathcal{A})} \times c(\mathcal{A})!$.

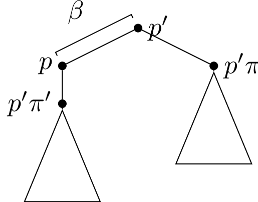



**Figure 2** Element of $\text{cr}(p)$.

Because $\text{cr}(p) \subseteq \mathcal{C}(\mathcal{A})$ for each $p \in \text{Pos}(\rho)$, there are at most $2^{c(\mathcal{A})}$ distinct sets $\text{cr}(p)$. Hence, we can extract a subsequence $v_0, \ldots, v_{k_2}$ of $u_1, \ldots, u_{k_1}$ such that $\text{cr}(v_0) = \ldots = \text{cr}(v_{k_2})$ and $k_2 = \frac{k_1}{2^{c(\mathcal{A})}} - 1 = (e \times k + 1) \times c(\mathcal{A})! - 1$.

The idea of the proof of Lemma 10 is illustrated in Figure 3. If for each $j = 1, \ldots, k$ the weak run $\rho[\rho|_{v_0}]_{v_j}$ has a close equality, then (for large enough $k_2$) there is a (long enough) subsequence $w_1, \ldots, w_m$ of $v_1, \ldots, v_{k_2}$ such that "the same" close equality is created in $\rho[\rho|_{v_0}]_{w_j}$ for each $j = 1, \ldots, m$. We recursively consider the sequence $w_1, \ldots, w_m$ – the number of possible places where a close equality may be created is now smaller – we eliminated one element of $\text{cr}(v_0) = \text{cr}(w_j)$. If $g(\mathcal{A}, k)$ is large enough then we will ultimately eliminate all possible elements of $\text{cr}(v_0)$. Then no close equality can be created in $\rho[\rho|_{v_0}]_{v_j}$ because for each element of $\text{cr}(v_0) = \text{cr}(v_j)$ the subterms at the corresponding positions below $v_0$ and $v_j$ are identical.

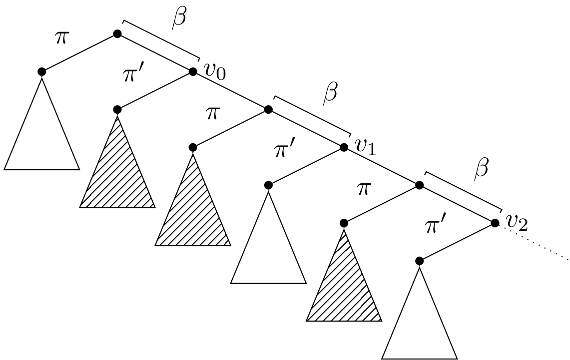



**Figure 3** The proof of the pumping lemma.

We proceed with a precise proof.

The *dependency degree* of a subsequence $v_{i_0}, \ldots, v_{i_m}$ is:

$$\text{dep}(v_{i_0} \ldots v_{i_m}) = |\{(\beta, \pi, \pi') \in \text{cr}(v_0) \mid t|_{(v_{i_0}/\beta)\pi} = \ldots = t|_{(v_{i_m}/\beta)\pi}\}|$$

where $t$ is the term associated to $\rho$.

Let $f(n)$ be the function recursively defined on the interval $[0 \ldots c(\mathcal{A})]$ by:

$$\begin{aligned} f(c(\mathcal{A})) &= k \\ f(n) &= (c(\mathcal{A}) - n) \times (f(n+1) + 1) + k - 1 \text{ for } n < c(\mathcal{A}) \end{aligned}$$

The next lemma is an analogon of Lemma 22 from [7]. It is the main technical lemma needed in the proof of Lemma 10.

▶ **Lemma 14.** *Assume*

($\star$) *for all* $0 \leq j \leq k_2$ *the cardinal of the set* $\{j' \mid k_2 \geq j' > j, \rho[\rho|_{v_j}]_{v_{j'}}$ *has no close equality* $\}$ *is smaller than* $k$.

*Then for all* $0 \leq n \leq c(\mathcal{A})$, *there exists a subsequence* $v_{i_0} \dots v_{i_{f(n)}}$ *of* $v_0 \dots v_{k_2}$ *such that* $\mathrm{dep}(v_{i_0} \dots v_{i_{f(n)}}) \geq n$.

**Proof.** The proof is an adaptation of the proof of Lemma 22 in [7], by induction on $n$.

The case $n = 0$ is exactly the same as in the proof of Lemma 22 in [7], showing $f(0) \leq k_2$. Let $F(n) = f(c(\mathcal{A}) - n)$ for all $0 \leq n \leq c(\mathcal{A})$. We have:

$$\begin{array}{rcll} F(0) & = & k & \\ F(n) & = & n(F(n-1)+1)+k-1 & \text{for } 1 \leq n \leq c(\mathcal{A}) \end{array}$$

Thus:

$$\begin{array}{rcl} F(n) & = & n! \times (F(0)+1) + k \times \sum_{i=1}^{n} \frac{1}{i!} - 1 \\ & \leq & k \times n! + n! + k \times n! \times (e-1) - 1 \\ & = & n! \times (k \times e + 1) - 1 \end{array}$$

Hence, $f(0) = F(c(\mathcal{A})) \leq c(\mathcal{A})! \times (k \times e + 1) - 1 = k_2$.

For $n+1$, we proceed analogously to [7]. Assume the property is true for $n < c(\mathcal{A})$. By the induction hypothesis, we have a subsequence $v_{i_0} \dots v_{i_{f(n)}}$ extracted from $v_0 \dots v_{k_2}$ such that $\mathrm{dep}(v_{i_0} \dots v_{i_{f(n)}}) \geq n$. By the assumption ($\star$), for at least $f(n) - (k-1) = (c(\mathcal{A}) - n) \times (f(n+1)+1) =: k_3$ positions $w$ among $v_{i_1} \dots v_{i_{f(n)}}$, the weak run $\rho[\rho|_{v_{i_0}}]_w$ has a close equality (close to $w$; we take $j = i_0$ in ($\star$) to conclude that there are at least $k_2 - i_0 - (k-1)$ indices $j'$ such that $\rho[\rho|_{v_{i_0}}]_{v_{j'}}$ has a close equality; now $k_2 - i_0 \geq f(n)$ because there exist $f(n)$ indices $i_0 < i_1 < \dots < i_{f(n)} \leq k_2$). Let $w_1 \dots w_{k_3}$ be a subsequence of $v_{i_1} \dots v_{i_{f(n)}}$ consisting of the positions $w$ as above, i.e., for all $j = 1, \dots, k_3$ the weak run $\rho[\rho|_{v_{i_0}}]_{w_j}$ has a close equality. Hence, for $j = 1, \dots, k_3$ there exists $(\beta_j, \pi_j, \pi'_j) \in \mathrm{cr}(w_j) = \mathrm{cr}(v_0)$ such that:

- $t|_{(v_{i_0}/\beta_j)\pi'_j} \neq t|_{(v_{i_0}/\beta_j)\pi_j}$,
- $t|_{(v_{i_0}/\beta_j)\pi'_j} = t|_{(w_j/\beta_j)\pi_j}$,

where $t$ is the term associated with $\rho$. Thus $t|_{(v_{i_0}/\beta_j)\pi_j} \neq t|_{(w_j/\beta_j)\pi_j}$ for $j = 1, \dots, k_3$.

Because $\mathrm{dep}(v_{i_0} \dots v_{i_{f(n)}}) \geq n$, there exists a subset $E \subseteq \mathrm{cr}(v_0)$ such that $|E| = n$ and $t|_{(v_{i_0}/\beta)\pi} = \dots = t|_{(v_{i_{f(n)}}/\beta)\pi}$ for $(\beta, \pi, \pi') \in E$. In particular, $t|_{(v_{i_0}/\beta)\pi} = t|_{(w_1/\beta)\pi} = \dots = t|_{(w_{k_3}/\beta)\pi}$ for $(\beta, \pi, \pi') \in E$. Hence, $\{(\beta_1, \pi_1, \pi'_1), \dots, (\beta_{k_3}, \pi_{k_3}, \pi'_{k_3})\} \cap E = \emptyset$ (because $t|_{(v_{i_0}/\beta_j)\pi_j} \neq t|_{(w_j/\beta_j)\pi'_j}$ for $j = 1, \dots, k_3$). By Fact 13 we have $|\mathrm{cr}(v_0)| \leq c(\mathcal{A})$. Thus, there are at most $c(\mathcal{A}) - n$ distinct tuples among $(\beta_1, \pi_1, \pi'_1), \dots, (\beta_{k_3}, \pi_{k_3}, \pi'_{k_3})$. Thus there exist $1 \leq j_0 < \dots < j_{f(n+1)} \leq k_3$ such that $(\beta_{j_0}, \pi_{j_0}, \pi'_{j_0}) = \dots = (\beta_{j_{f(n+1)}}, \pi_{j_{f(n+1)}}, \pi'_{j_{f(n+1)}})$, because $\frac{k_3}{c(\mathcal{A})-n} = f(n+1)+1$. Let $(\beta', \pi, \pi') = (\beta_{j_0}, \pi_{j_0}, \pi'_{j_0})$ be this tuple. Because $t|_{(w_j/\beta')\pi} = t|_{(v_{i_0}/\beta')\pi}$ for $1 \leq j \leq k_3$, $t|_{(w_{j_0}/\beta')\pi} = \dots = t|_{(w_{j_{f(n+1)}}/\beta')\pi}$. Since $(\beta', \pi, \pi') \notin E$:

$$\mathrm{dep}(w_{j_0} \dots w_{j_{f(n+1)}}) > \mathrm{dep}(v_{i_0} \dots v_{i_{f(n)}}) \geq n.$$

This completes the proof because $w_{j_0} \dots w_{j_{f(n+1)}}$ is a subsequence of $v_0 \dots v_{k_2}$. ◀

**Proof of Lemma 10.** Follows the proof of Lemma 19 in [7]. Assume ($\star$) holds to derive a contradiction. Then for $n = c(\mathcal{A})$ and $f(n) = k$ there exists a subsequence $v_{i_0} \dots v_{i_k}$ of $v_0 \dots v_{k_2}$ such that $\mathrm{dep}(v_{i_0} \dots v_{i_k}) \geq c(\mathcal{A})$. But $|\mathrm{cr}(v_0)| \leq c(\mathcal{A})$ by Fact 13, so for all $(\beta, \pi, \pi') \in \mathrm{cr}(v_0)$ we have $t|_{(v_{i_0}/\beta)\pi} = \dots = t|_{(v_{i_k}/\beta)\pi}$.

Assume $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ has a close equality for some $1 \le j \le k$. There is $(\beta, \pi, \pi') \in \mathrm{cr}(v_{i_j}) = \mathrm{cr}(v_0)$ such that $t|_{(v_{i_0}/\beta)\pi'} \ne t|_{(v_{i_0}/\beta)\pi}$ and $t|_{(v_{i_0}/\beta)\pi'} = t|_{(v_{i_j}/\beta)\pi}$. Hence, $t|_{(v_{i_0}/\beta)\pi} \ne t|_{(v_{i_j}/\beta)\pi}$. Contradiction. Thus, each $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ has no close equality for $1 \le j \le k$. Then the cardinality of the set in ($\star$) for $j = 0$ is at least $k$ (note that by definition $k \le k_2$) which contradicts ($\star$).

Thus, ($\star$) cannot hold. This implies that for $1 \le j \le k$ the upward pumping $\rho[\rho|_{v_{i_0}}]_{v_{i_j}}$ does not have a close equality. ◀

▶ **Corollary 15.** *Let $\rho$ be a run of $\mathcal{A}$ and $p_1 \prec \ldots \prec p_{g(\mathcal{A},k)}$ be positions of $\rho$ such that $|p_{j+1}/p_j| > d(\mathcal{A})$ (i.e., the distance between two consecutive positions greater than $d(\mathcal{A})$). Then there exist indices $i_0 < \ldots < i_k$ such that the pumping $\rho[\rho_j^m]_{p_{i_j}}$ for $j > 0$ does not have a close equality for any $m \ge 0$ where: $\rho_j^0 = \rho|_{p_{i_0}}$ and $\rho_j^{m+1} = \rho|_{p_{i_0}}[\rho_j^m]_{p_{i_j}/p_{i_0}}$. See Figure 4.*

**Proof.** Since $p_1 \prec \ldots \prec p_{g(\mathcal{A},k)}$, we have $\rho|_{p_1} \ggg \ldots \ggg \rho|_{p_{g(\mathcal{A},k)}}$. By Lemma 10 there exist indices $i_0 < \ldots < i_k$ such that $\rho[\rho_j^0]_{p_{i_j}}$ does not have a close equality. Since $|p_{i_j}/p_{i_0}| > d(\mathcal{A})$, noting that $\rho[\rho_j^0]_{p_{i_j}} = \rho[\rho|_{p_{i_0}}[\rho_j^0]_{p_{i_j}/p_{i_0}}]_{p_{i_0}}$, we can prove by induction on $m$ that $\rho[\rho_j^m]_{p_{i_j}}$ has no close equality either. Indeed, any close equality in $\rho[\rho_j^{m+1}]_{p_{i_j}}$ must be a close equality in $\rho|_{p_{i_0}}[\rho_j^m]_{p_{i_j}/p_{i_0}}$, because $|p_{i_j}/p_{i_0}| \ge d(\mathcal{A})$. But then we would have the same close equality in $\rho|_{p_{i_0}}[\rho_j^0]_{p_{i_j}/p_{i_0}}$. ◀

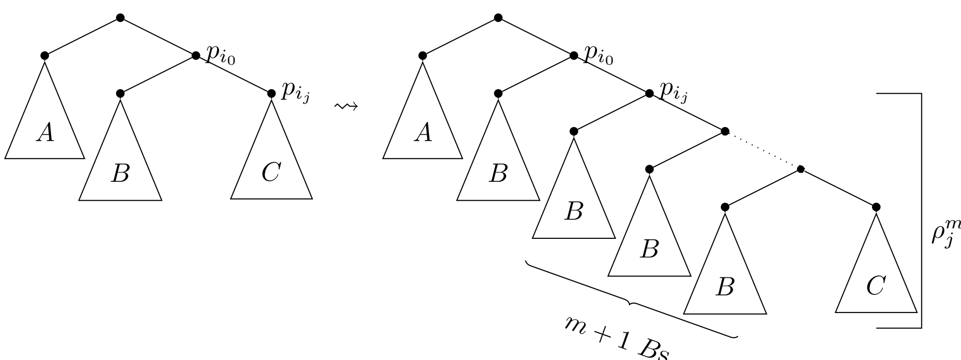



**Figure 4** Repeated pumping.

▶ **Corollary 16.** *Under the assumptions of the previous Corollary 15, there exist indices $i_0 < \ldots < i_k$ and $m_0 \ge 0$ such that the pumping $\rho[\rho_j^m]_{p_{i_j}}$ for $j > 0$ and $m \ge m_0$ does not have any equality (close or far).*

**Proof.** By Corollary 15, $\rho[\rho_j^m]_{p_{i_j}}$ does not have a close equality for any $m \ge 0$. We can choose $m$ to be large enough so that no far equality is created either. Indeed, if an equality $(p, \pi, \pi')$ far from $p_{i_j}$ is created, then e.g. $p\pi \preceq p_{i_j}$ and $p\pi' \parallel p_{i_j}$ and $t|_{p\pi} = t|_{p\pi'}$. By making $m$ large enough we can ensure $|\rho_j^m| > |t|_{p'}|$ for any $p' \parallel p_{i_j}$, and then the equality $t|_{p\pi} = t|_{p\pi'}$ is impossible. ◀

**Proof of the Bound Theorem 7.** If the height of the run is $\ge G(\mathcal{A})$ then we can choose $g(\mathcal{A}, 1)$ positions $p_1, \ldots, p_{g(\mathcal{A},1)}$ satisfying the requirements of Corollary 16. This gives us infinitely many different accepting runs $\rho[\rho_j^m]_{p_{i_j}}$ for $m \ge m_0$. Conversely, if the language is infinite then there can be no bound on the maximal height of an accepting run. ◀

# 5 Automaton recognising the intersection of a regular tree language with the set of normal forms of a TRS

Given a tree grammar $G$ it is standard to construct a finite tree automaton $\mathcal{A}_G$ recognising the language $L(G)$. See e.g. [5].

The next step is to construct the normal forms ADC $\mathcal{A}_R$ for a given term rewriting system $R$. The automaton $\mathcal{A}_R$ recognises the ground normal forms of $R$. The constraints are necessary to handle non-left-linear rules in $R$. No constraints are generated if $R$ is left-linear.

Finally, we construct the product automaton $\mathcal{A}_G \times \mathcal{A}_R$ which recognises the intersection of $L(\mathcal{A}_G)$ and $L(\mathcal{A}_R)$.

## 5.1 Construction of the normal forms automaton

The construction of $\mathcal{A}_R$ is described in detail in [7]. We recall it for completeness.

- Let $\mathcal{L}$ be the set of the left-hand sides of $R$.
- Let $\mathcal{L}_1$ be the subset of the linear terms in $\mathcal{L}$.
- Let $\mathcal{L}_2$ be the set of linearisations of the nonlinear terms in $\mathcal{L}$. For each $l \in \mathcal{L}_2$ we denote its nonlinear origin by $\#l \in \mathcal{L}$.
- Let $Q_0$ consist of all strict subterms of terms in $\mathcal{L}_1 \cup \mathcal{L}_2$ (modulo renaming of variables) plus two special states:
  - a single variable $x$ which will accept all terms,
  - $q_r$ which will accept only reducible terms of $R$.

  Note $|Q_0| \leq ||R|| + 2$.
- The set of states $Q_R$ consists of all unifiable subsets of $Q_0 \setminus \{q_r\}$ plus $q_r$. Each element of $Q_R$ different from $q_r$ is denoted by $q_u$ where $u$ is the term resulting from unifying all elements of the state with the mgu of the state. Note $|Q_R| \leq 2^{|Q_0|} \leq 2^{||R||+2}$.
- $\Delta_R$ is the set of all rules of the form

  $$f(q_{u_1}, \ldots, q_{u_n}) \xrightarrow{c} q_u$$

  where $q_{u_1}, \ldots, q_{u_n}, q_u \in Q_R$ and:
  1. if one of the $q_{u_i}$'s is $q_r$ or $f(u_1, \ldots, u_n)$ is an instance of some $s \in \mathcal{L}_1$, then $q_u = q_r$ and $c = \top$,
  2. otherwise, $u$ is the mgu of all terms $v \in Q_0 \setminus \{q_r\}$ such that $f(u_1, \ldots, u_n)$ is an instance of $v$, and the constraint $c$ is defined by:

  $$\bigwedge_{\substack{l \in \mathcal{L}_2 \\ u, l \text{ unifiable} \\ f(u_1,\ldots,u_n) \text{ is an instance of } l}} \quad \bigvee_{\substack{x \in \mathrm{Var}(\#l) \\ \#l|_{p_1} = \#l|_{p_2} = x \\ p_1 \neq p_2}} p_1 \neq p_2$$
- Take $\mathcal{A}_R = (Q_R, Q_R \setminus \{q_r\}, \Delta_R)$.

$|Q_R|$ is exponential in $R$ and each constraint has size polynomial in $||R||$.

## 5.2 Construction of the product automaton

Given two ADCs $\mathcal{A}_1 = (Q_1, Q_1^f, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, Q_2^f, \Delta_2)$, the *product ADC* $\mathcal{A}_1 \times \mathcal{A}_2 = (Q, Q^f, \Delta)$ is defined by:

- $Q = Q_1 \times Q_2$,
- $Q^f = Q_1^f \times Q_2^f$,
- $\Delta$ consists of the transitions $f((q_1, q_1'), \ldots, (q_n, q_n')) \xrightarrow{c_1 \wedge c_2} (q, q')$ for every pair of transitions $f(q_1, \ldots, q_n) \xrightarrow{c_1} q \in \Delta_1$ and $f(q_1', \ldots, q_n') \xrightarrow{c_2} q' \in \Delta_2$.

Note that $|Q| = |Q_1| \times |Q_2|$, $|\Delta| = |\Delta_1| \times |\Delta_2|$, $c(\mathcal{A}) \leq c(\mathcal{A}_1) + c(\mathcal{A}_2)$, $s(\mathcal{A}) \leq s(\mathcal{A}_1) + s(\mathcal{A}_2)$, $n(\mathcal{A}) \leq n(\mathcal{A}_1) + n(\mathcal{A}_2)$ and $d(\mathcal{A}) = \max(d(\mathcal{A}_1), d(\mathcal{A}_2))$ (there are no new atomic constraints).

## 6 Emptiness

To check if $L(G) \cap \mathrm{NF}(R) = \emptyset$, we run the emptiness decision algorithm from [7] on $\mathcal{A}_G \times \mathcal{A}_R$. The algorithm runs in exponential time. For completeness, we give a brief presentation of the emptiness decision algorithm from [7]. The following lemmas and definitions come from [7]. Let $\mathcal{A} = (Q, Q_f, \Delta)$ be the ADC whose emptiness we want to check.

▶ **Definition 17.** *The ordering $\gg$ on terms over $\Delta$ is defined by: $\rho_1 \gg \rho_2$ iff $I(\rho_1) > I(\rho_2)$ where $I(\rho)$ is the triple $(\mathrm{depth}(\rho), M(\rho), \rho)$ with $M(\rho)$ the multiset of strict subterms of $\rho$. The ordering $>$ on triples is the lexicographic product of:*
1. *the ordering on natural numbers,*
2. *the multiset extension of $\gg$ (see e.g. [1, Definition 2.5.3]),*
3. *the lexicographic path order extending a total order on the signature (see e.g. [1, Definition 5.4.12]).*

The lexicographic path order in the third component may be replaced by any reduction order total on ground terms.

▶ **Lemma 18.** *$\gg$ is monotonic, well-founded and total on ground terms. Moreover, if $\mathrm{depth}(\rho) > \mathrm{depth}(\rho')$ then $\rho \gg \rho'$.*

One could replace $\gg$ with any order satisfying the conditions of the above lemma.

▶ **Definition 19** (Emptiness decision algorithm). *Let $E_q^0 = \emptyset$ for each state $q \in Q$. For $m \geq 0$, let $E_q^{m+1}$ consist of all runs $\rho = r(\rho_1, \ldots, \rho_n)$ such that:*
- *$\rho_1, \ldots, \rho_n \in \bigcup_{i=0}^{m} \bigcup_{q \in Q} E_q^i$,*
- *the target state of $\rho$ is $q$,*
- *for every $p \in \mathrm{Pos}(\rho) \setminus \mathcal{S}(\mathcal{A})$ with $|p| \leq d(\mathcal{A}) + 1$, there is no sequence of length $b(\mathcal{A})$ of runs $\rho_1', \ldots, \rho_{b(\mathcal{A})}'$ in $\bigcup_{i=0}^{m} \bigcup_{q \in Q} E_q^i$ such that $\rho|_p \gg \rho_{b(\mathcal{A})}' \gg \ldots \gg \rho_1'$ and $\rho(p)$,$\rho_1'(\epsilon)$,…,$\rho_{b(\mathcal{A})}'(\epsilon)$ all have the same target state and for every $1 \leq j \leq b(\mathcal{A})$ the pumping $\rho[\rho_j']_p$ does not contain any equality close to $p$.*

*After a finite number of iterations, we obtain the saturated set $E^* = \bigcup_{m \geq 0} \bigcup_{q \in Q} E_q^m$. The language of $\mathcal{A}$ is empty iff $E^*$ does not contain an accepting run.*

A more detailed pseudocode of the algorithm and the calculation of $b(\mathcal{A})$ may be found in Appendix A. The correctness of the algorithm is proven in [7].

▶ **Theorem 20.** *The emptiness decision algorithm runs in time $O(|\mathcal{A}|^{P_0(s(\mathcal{A}))})$ where $P_0$ is a polynomial.*

**Proof.** In [7, Theorem 28] it is shown that the emptiness decision algorithm runs in time $O((|Q| \times |\Delta|)^{P_0'(\mathrm{cs}(\mathcal{A}))})$ where $P_0'$ is a polynomial and $\mathrm{cs}(\mathcal{A})$ is the total size of all constraints in $\mathcal{A}$. A careful analysis of the bounds in Lemma 27 and Sections 5.3.2, 5.3.3 in [7] reveals that the exponent in the running time can actually be made polynomial in $s(\mathcal{A})$ at the

expense of introducing the size of the automation $|\mathcal{A}|$ into the base. More precisely, the inequalities in Lemma 27 and Sections 5.3.2 and 5.3.3 in [7] initially the exponent is bounded by a polynomial in $s(\mathcal{A})$ (denoted $c(\mathcal{A})$ in [7]), and only this exponent is then replaced according to the inequality $s(\mathcal{A}) \le d(\mathcal{A})n(\mathcal{A})$. Instead, one can take a smaller bound on $h(\mathcal{A}, k)$ in Lemma 27, then plug it into the inequalities in Section 5.3.2 to get a bound with an exponent polynomial in $s(\mathcal{A})$ and a different base. The base needs to be $|\mathcal{A}|$ instead of $|Q| \times |\Delta|$, because without $\mathrm{cs}(\mathcal{A})$ in the exponent one cannot remove the $\mathrm{cs}(\mathcal{A})$ factor from the base. ◀

In the above theorem, we need $s(\mathcal{A})$ in the exponent instead of $\mathrm{cs}(\mathcal{A})$ because of the product automaton construction: we have $s(\mathcal{A}_1 \times \mathcal{A}_2) \le s(\mathcal{A}_1) + s(\mathcal{A}_2)$, but this inequality does not hold for cs. In particular, $\mathrm{cs}(\mathcal{A}_G \times \mathcal{A}_R) = |\Delta_{\mathcal{A}_G}| \times \mathrm{cs}(\mathcal{A}_R)$ while $s(\mathcal{A}_G \times \mathcal{A}_R) = s(\mathcal{A}_R)$.

▶ **Proposition 21** ([5, Theorem 1.7.5]). *The following problem is EXPTIME-hard: given $n$ tree automata $\mathcal{A}_1, \dots, \mathcal{A}_n$, is $L(\mathcal{A}_1) \cap \dots \cap L(\mathcal{A}_n)$ empty?*

Note that $n$ is a part of the input, *not* a constant.

▶ **Theorem 22.** *Given $n$ finite tree automata $\mathcal{A}_1, \dots, \mathcal{A}_n$, there is a polynomial-time construction of a linear term rewriting system $R$ such that:*

$$\mathrm{NF}(R) = \{g(s) \mid s \text{ encodes accepting runs of } \mathcal{A}_1, \dots, \mathcal{A}_n \text{ on a common term}\}$$

*The encoding is such that for each $n$-tuple of runs there exists exactly one term representing this tuple of runs. In particular:*

- $\mathrm{NF}(R) = \emptyset$ *iff* $L(\mathcal{A}_1) \cap \dots \cap L(\mathcal{A}_n) = \emptyset$,
- $\mathrm{NF}(R)$ *is finite iff* $L(\mathcal{A}_1) \cap \dots \cap L(\mathcal{A}_n)$ *is finite.*

**Proof.** The construction of the term rewriting system $R$ is exactly the one from [7, Section 6]. We refer there for details. The statement concerning finiteness (the second point) is not present in [7], but it is easily checked. ◀

▶ **Theorem 23.** *Given a regular tree grammar $G$ and a term rewriting system $R$, the problem of checking the emptiness of $L(G) \cap \mathrm{NF}(R)$ is EXPTIME-complete. The problem is EXPTIME-hard already for linear $R$.*

**Proof.** To decide emptiness of intersection, we construct a finite tree automaton (i.e. an ADC without constraints) $\mathcal{A}_G$ with $L(\mathcal{A}_G) = L(G)$, and the normal forms ADC $\mathcal{A}_R$. Then we check the emptiness of the product $\mathcal{A}_G \times \mathcal{A}_R$. We have $|\mathcal{A}_G| = O(|G|)$ and $|\mathcal{A}_R| = O(2^{||R||})$ and $s(\mathcal{A}_R) = O(P_1(|R|))$ for some polynomial $P_1$. Then $|\mathcal{A}_G \times \mathcal{A}_R| = O(|G|2^{||R||})$ and $s(\mathcal{A}_G \times \mathcal{A}_R) \le s(\mathcal{A}_G) + s(\mathcal{A}_R) = s(\mathcal{A}_R) = O(P_1(||R||))$. Constructing $\mathcal{A}_G \times \mathcal{A}_R$ takes time proportional to $|\mathcal{A}_G \times \mathcal{A}_R|$. Hence, by Theorem 20 the entire procedure takes time $O((|G|2^{||R||})^{P_0(P_1(||R||))}) = O(|G|^{P(||R||)})$ for some polynomial $P$.

EXPTIME-hardness follows from Proposition 21, taking $G$ with $L(G) = \mathcal{T}(\mathcal{F})$ (the set of all ground terms) and the $R$ constructed in Theorem 22. ◀

## 7 Finiteness

By adapting the arguments of [7] for downward pumping, the Bound Theorem 7 could be refined to provide, in addition to the lower bound, also an exponential upper bound on the height of an accepted term. Then a direct application of the Bound Theorem would yield a 3-EXPTIME algorithm for deciding finiteness: check if there are any terms with height between the two bounds. Instead, we use the Bound Theorem together with the emptiness decision algorithm for ADCs to show that the finiteness problem is in EXPTIME.

▶ **Definition 24.** *For a given $N \in \mathbf{N}$, we define an automaton $\mathcal{A}_N = (Q_N, Q_N^f, \Delta_N)$ recognising the language of all terms of height at least $N$.*

- $Q_N = \{q_i \mid i \in \{0, \ldots, N\}\}$,
- $Q_N^f = \{q_N\}$,
- $\Delta_N$ *consists of the transitions:*
  - $a \to q_0$,
  - $f(q_{i_1}, \ldots, q_{i_n}) \to q_{\min(\max(i_1,\ldots,i_n)+1,N)}$ *for $n > 0$ and all $i_1, \ldots, i_n \in \{0, \ldots, N\}$.*

*Intuitively, state $q_i$ indicates that a subterm has height at least $i$.*

▶ **Theorem 25.** *Assume the maximum function symbol arity is a fixed constant. Given a regular tree grammar $G$ and a term rewriting system $R$, the problem of checking the finiteness of $L(G) \cap \mathrm{NF}(R)$ is EXPTIME-complete. The problem is EXPTIME-hard already for linear $R$.*

**Proof.** To decide finiteness in exponential time, we first construct the automaton $\mathcal{A} = \mathcal{A}_G \times \mathcal{A}_R$ like in Theorem 23. Then take $\mathcal{A}' = \mathcal{A} \times \mathcal{A}_N$ with

$$N = H(\mathcal{A}) = (e + 1) \times |Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A}) + 1)$$

where $H(\mathcal{A})$ is the function from Theorem 7 and $\mathcal{A}_N$ is the automaton from Definition 24 recognising the language of all terms of height at least $N$. The language of $\mathcal{A}'$ consists of all terms in $L(G) \cap \mathrm{NF}(R)$ with height at least $N$. By Theorem 7 the language $L(\mathcal{A}) = L(G) \cap \mathrm{NF}(R)$ is finite iff all terms accepted by $\mathcal{A}$ have height $< N$. Hence, $L(\mathcal{A}') = \emptyset$ iff $L(G) \cap \mathrm{NF}(R)$ is finite. Thus, it suffices to check emptiness of $\mathcal{A}'$ with the algorithm outlined in the previous section.

By the proof of Theorem 23 we have $|\mathcal{A}| = O(|G|2^{||R||})$. Since $|\mathcal{A}_N| = O(N^\alpha)$ with $\alpha$ a constant depending on the maximum function symbol arity, we obtain $|\mathcal{A}'| = O(|G|2^{||R||}N^\alpha)$. Also $s(\mathcal{A}') = s(\mathcal{A}) = O(P_1(||R||))$. Hence, by Theorem 20 running the emptiness decision algorithm on $\mathcal{A}$ takes time:

$$\begin{aligned}
&O\left((|G|2^{||R||}N^\alpha)^{P_0(P_1(||R||))}\right) = \\
&O(|G|^{P_2(||R||)}N^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)}(|Q| \times 2^{c(\mathcal{A})} \times c(\mathcal{A})! \times (d(\mathcal{A}) + 1))^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)}(|G|2^{||R||} \times 2^{||R||} \times 2^{||R|| \log(||R||)} \times ||R||)^{P_3(||R||)}) = \\
&O(|G|^{P_2(||R||)}(|G|2^{P_4(||R||)})^{P_3(||R||)}) = \\
&O(|G|^{P(||R||)})
\end{aligned}$$

where the polynomial $P$ depends on the maximum function symbol arity.

To show EXPTIME-hardness, we reduce from the problem of the finiteness of the intersection of the languages of $n$ tree automata: given $n$ tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, is $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ finite? The reduction follows directly from Theorem 22 (taking $G$ with $L(G) = \mathcal{T}(\mathcal{F})$). It remains to show that the finiteness problem for the intersection of the languages of $n$ tree automata is EXPTIME-hard. We reduce the problem of emptiness of intersection of $n$ tree languages (see Proposition 21).

For an automaton $\mathcal{A} = (Q, Q_f, \Delta)$ over signature $\Sigma$ we create an automaton $\mathcal{A}' = (Q', Q'_f, \Delta')$ over $\Sigma'$ such that $L(\mathcal{A})$ is empty iff $L(\mathcal{A}')$ is finite. Each non-nullary symbol $f \in \Sigma$ is in $\Sigma'$. For each constant $c \in \Sigma$ we have a unary symbol $c \in \Sigma'$. There is an extra unary symbol $S \in \Sigma' \setminus \Sigma$ and an extra constant $C \in \Sigma' \setminus \Sigma$. We set $Q' = Q \cup \{q_S\}$ and $Q'_f = Q_f$. The transitions $\Delta'$ include:

- $f(q_1, \ldots, q_n) \to q$ if it is in $\Delta$ and $n > 0$,
- $c(q_S) \to q$ if $c \to q \in \Delta$,
- $C \to q_S$,
- $S(q_S) \to q_S$.

The automaton $\mathcal{A}$ accepts a term $t$ iff $\mathcal{A}'$ accepts terms which result from $t$ by replacing each constant occurrence $c$ with $c(S^k(C))$ for some $k$ (possibly different $k$ for different occurrences). Now, $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ is finite iff $L(\mathcal{A}'_1) \cap \ldots \cap L(\mathcal{A}'_n)$ is finite. Indeed $L(\mathcal{A}'_1) \cap \ldots \cap L(\mathcal{A}'_n)$ contains all terms from $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ with each constant $c$ replaced with $c(S^k(C))$ for some $k$. ◀

## 8 Experiments

We evaluate our approach using three examples. The first example is a minimal example inspired by Boolean algebra and highlights the limitations of the approach, as well as some opportunities to overcome them. Examples 2 and 3 extend practical examples from the literature [11]: Example 2 applies the technique to the automatic construction of programs. Example 3 computes paths through a large labyrinth in order to illustrate scalability with linear rewrite systems compared to the SMT-solver based approach in [11]. All examples are available in our Haskell implementation, which accompanies this paper [2].

### 8.1 Example 1 - Boolean Algebra

Single-sorted Boolean ground terms over a signature containing a binary function symbol **AND** and constants **T**, **F** are recognised by the tree grammar $G_B$:

$$G_B \quad = \quad (b, \{b\}, \{\mathbf{T}, \mathbf{F}, \mathbf{AND}\}, \{b \to \mathbf{T},\ b \to \mathbf{F},\ b \to \mathbf{AND}(b, b)\})$$

A simple rewrite system can normalize terms by evaluating all function applications of **AND**. One way to specify evaluation rules for **AND** is to use the rewrite system $RS_B$:

$$RS_B \quad = \quad \{\mathbf{AND}(\mathbf{F}, x) \to \mathbf{F},\ \mathbf{AND}(x, \mathbf{F}) \to \mathbf{F},\ \mathbf{AND}(x, x) \to x\}$$

Using the construction in Section 5.1 yields the normal forms ADC $\mathcal{A}_B = (Q_B, Q_B^f, \Delta_B)$ recognizing $\mathrm{NF}(RS_B)$:

$$\begin{aligned} Q_B \quad &= \quad \{q_0, q_1, q_2\} \qquad Q_B^f = \{q_1, q_2\} \\ \Delta_B \quad &= \quad \{\mathbf{T} \xrightarrow{\top} q_1,\ \mathbf{F} \xrightarrow{\top} q_2,\ \mathbf{AND}(q_1, q_1) \xrightarrow{1 \neq 2} q_1\} \\ &\quad\ \cup \{\mathbf{AND}(p_1, p_2) \xrightarrow{\top} q_0 \mid p_1, p_2 \in Q, p_1 \neq q_1 \lor p_2 \neq q_1\} \end{aligned}$$

The language $L(G_B) \cap L(\mathcal{A}_B) = \{\mathbf{T}, \mathbf{F}\}$ is finite and non-empty. In the worst case, the emptiness checking algorithm from Definition 19 needs to enumerate and store at least $b$ terms (if the result is empty), where $b$ is the value computed in Appendix A. For our example, the corresponding values are $b(\mathcal{A}_G \times \mathcal{A}_B) = b_{\text{empty}} = 235018$ for emptiness and $b(\mathcal{A}_G \times \mathcal{A}_B \times \mathcal{A}_N) = b_{\text{fin}} = 7300813834$ for finiteness. Here, the enumeration stops after the first iteration because there exists a term in $L(G_B) \cap L(\mathcal{A}_B)$ of height one. Since $L(G_B) \cap L(\mathcal{A}_B)$ is finite, the finiteness check must enumerate at least $b_{\text{fin}}$ terms, which is not practically feasible.

Manual inspection of our example reveals that the rewrite rule $\mathbf{AND}(x, x) \to x$ can be simplified to $\mathbf{AND}(\mathbf{T}, \mathbf{T}) \to \mathbf{T}$, while retaining the same set of normal forms.

$$RS_B^{\text{lin}} \quad = \quad \{\mathbf{AND}(\mathbf{F}, x) \to \mathbf{F},\ \mathbf{AND}(x, \mathbf{F}) \to \mathbf{F},\ \mathbf{AND}(\mathbf{T}, \mathbf{T}) \to \mathbf{T}\}$$

Using this simplification we obtain the ADC $\mathcal{A}_B^{\text{lin}} = (Q_B^{\text{lin}}, Q_B^{f,\text{lin}}, \Delta_B^{\text{lin}})$:

$$\begin{array}{rcl} Q_B^{\text{lin}} & = & \{q_0, q_1, q_2\} \qquad Q_B^{f,\text{lin}} = \{q_1, q_2\} \\ \Delta_B & = & \{\mathbf{T} \xrightarrow{\top} q_1,\ \mathbf{F} \xrightarrow{\top} q_2\} \cup \{\mathbf{AND}\ p \xrightarrow{\top} q_0 \mid p \in Q \times Q)\} \end{array}$$

The automaton $\mathcal{A}_B^{\text{lin}}$ is built for the linear rewrite system $RS_B^{\text{lin}}$ and all its disequality constraints are empty (true) by construction, resulting in a finite tree automaton without constraints. Hence, finiteness can be checked in polynomial time wrt. the automaton's size.

Our Haskell implementation exactly matches the expectations from theory: emptiness results are computed immediately (under 1 second on a laptop from 2018 with a 2,7 GHz quad core processor and 16GB Ram). For finiteness we had to abort after over 6 hours in the nonlinear case, while the linear case also computes in under 1 second.

## 8.2 Example 2 - Construction of sorting functions

Kallat et al. [11] describe how to perform program construction of applications of sorting functions using a tree grammar as an intermediate result of a type inhabitation algorithm. Their grammar (up to renaming of non-terminals) is given as follows:

$$\begin{array}{rcl} G_{\text{sort}} & = & (2, \{0,1,2,3,4\}, \{\mathbf{values}, \mathbf{id}, \mathbf{inv}, \mathbf{sortmap}, \mathbf{min}, \mathbf{default}, \mathbf{@}\}, \\ & & \{\ 4 \to \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, 1), 3),\ 2 \to \mathbf{@}(\mathbf{id}, 2),\ 2 \to \mathbf{@}(\mathbf{@}(\mathbf{min}, 0), 4), \\ & & 0 \to \mathbf{@}(\mathbf{id}, 0),\ 0 \to \mathbf{default},\ 0 \to \mathbf{@}(\mathbf{inv}, 0),\ 0 \to \mathbf{@}(\mathbf{@}(\mathbf{min}, 0), 4), \\ & & 1 \to \mathbf{id},\ 1 \to \mathbf{inv},\ 3 \to \mathbf{@}(\mathbf{id}, 3),\ 3 \to \mathbf{values}\}) \end{array}$$

Evaluation rules can be stated as the following rewrite system:

$$\begin{array}{rcl} RS_{\text{sort}} & = & \{\mathbf{@}(\mathbf{id}, x) \to x,\ \mathbf{@}(\mathbf{inv}, \mathbf{@}(\mathbf{inv}, x)) \to x, \\ & & \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, x), \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, y), z)) \to \mathbf{@}(\mathbf{@}(\mathbf{sortmap}, x), z) \\ & & \mathbf{@}(\mathbf{@}(\mathbf{min}, \mathbf{@}(\mathbf{@}(\mathbf{min}, x), y)), y) \to \mathbf{@}(\mathbf{@}(\mathbf{min}, x), y)\} \end{array}$$

The normal form ADC $\mathcal{A}_{\text{sort}}$ has 26 reachable states and 47 transitions (after reduction of non-reachable states). We obtain bound values $b(\mathcal{A}_G \times \mathcal{A}_{RS}) = b_{\text{empty}} = 4655986860$ and $b(\mathcal{A}_G \times \mathcal{A}_{RS} \times \mathcal{A}_N) = b_{\text{fin}} = 445281079421917788$. The language $L(G_{\text{sort}}) \cap L(\mathcal{A}_{\text{sort}})$ is finite and non-empty. Since the smallest term has a height of four, the emptiness checking algorithm terminates after four iterations with result *False* (non-empty). The algorithm for deciding finiteness needs to enumerate and store at least $b_{\text{fin}}$ terms before terminating with result *True* (finite).

In [11] no rewrite rules are used. Instead, the authors construct constraints that forbid terms of form $@(\mathbf{id}, x)$, $@(\mathbf{inv}, x)$, and $@(\mathbf{min}, @(x, y))$, declaring these forms as non-normal without providing replacements (i.e. the right-hand sides of the rewrite rules). Our approach is flexible enough to do the same, since right-hand sides of the rewrite system are ignored. We may use the following linear rules:

$$RS_{\text{sort}}^{\text{lin}} \quad = \quad \{@(\mathbf{id}, x) \to x,\ @(\mathbf{inv}, x) \to x,\ @(\mathbf{min}, @(x, y)) \to x\}$$

The normal forms automaton $\mathcal{A}_{\text{sort}}^{\text{lin}}$ is a finite tree automaton (no disequality constraints). Our Haskell implementation can check emptiness immediately and finiteness again is only possible in the linear case, with results being available in under one second.

## 8.3 Example 3 - Filtering redundant paths in a labyrinth

The last example in [11] is a grammar for finding paths through a labyrinth. In this grammar non-terminals are **up**, **down**, **left**, **right**, **start**. Rules ensure that only valid paths can be constructed. The example is scaled for randomly generated labyrinths. Redundant paths, such as $\mathbf{up}(\mathbf{down}(x))$ get filtered. The authors of [11] note that their SMT-solver based approach only scales up to labyrinths with $10 \times 10$ fields. Reproducing these experiments, we ran the CLS framework to generate labyrinth solution grammars for up to $30 \times 30$ fields (stopping there to limit the runtime of CLS). Using the four rewrite rules

$$RS_{\text{lab}} = \{\mathbf{up}(\mathbf{down}(x)) \to x,\ \mathbf{down}(\mathbf{up}(x)) \to x,\ \mathbf{left}(\mathbf{right}(x)) \to x,\ \mathbf{right}(\mathbf{left}(x)) \to x\}$$

our Haskell implementation again produces immediate emptiness results for the intersection, while finiteness is computed in under 5 minutes. The reduced intersection automaton has 7619 reachable states, 8956 transitions and a value $b(\mathcal{A}_G \times \mathcal{A}_{RS} \times \mathcal{A}_N) = b_{\text{fin}} = 149360346492$. This result is a true improvement over scalability issues encountered in solver-based solutions.

# 9 Conclusion

We have shown that the emptiness and finiteness problems of the intersection $L(G) \cap \text{NF}(R)$ of the language of a regular tree grammar $G$ and the normal forms of a rewrite system $R$ are EXPTIME-complete. Both problems are practically relevant for enumerating terms generated by the CLS synthesis algorithm (and potentially other synthesis approaches). Enumeration can be implemented by bottom-up enumerating all terms of $L(G)$ and filtering them according to membership in $\text{NF}(R)$. Without the decision procedures the enumeration algorithm does not know when to (not) stop: in the empty case it does not need to enumerate anything. In the finite case, it needs to enumerate until all terms of height $N$ (as computed in the proof of Theorem 25) are listed. In the infinite case, it can continue to enumerate.

We have also conducted practical experiments, which show that, although also EXPTIME-complete, the problems are feasible for left-linear rewrite systems. Results for the nonlinear case, however, were less encouraging, since here the proposed algorithm always has to enumerate a very large set of terms before being able to decide emptiness and an even larger set before being able to decide finiteness. It is an interesting goal for future research to investigate other algorithms, which might perform better in the average case. Also, heuristics that are incomplete (e.g., with bounds on the probability of obtaining a decision) are an interesting area for future research.

## References


1 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

2 J. Bessai, L. Czajka, F. Laarmann, and J. Rehof. Restricting tree grammars with rewriting. `https://github.com/FelixLaarmann/tree-grammar`, 2022.

3 Jan Bessai. *A type-theoretic framework for software component synthesis*. PhD thesis, Technical University of Dortmund, Germany, 2019. URL: `http://hdl.handle.net/2003/38387`.

4 Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro, and Jakob Rehof. Mixin composition synthesis based on intersection types. *Logical Methods in Computer Science*, 14(1), 2018. `doi:10.23638/LMCS-14(1:18)2018`.

5 Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. INRIA, 2008.

**6** Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. In *LICS 1997*, pages 26–34, 1997.

**7** Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. *Inf. Comput.*, 187(1):123–153, 2003.

**8** Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Bounded combinatory logic. 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 243–258. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.CSL.2012.243.

**9** Guillem Godoy and Omer Giménez. The HOM problem is decidable. *J. ACM*, 60(4):23:1–23:44, 2013.

**10** Fadil Kallat, Carina Mieth, Jakob Rehof, and Anne Meyer. Using component-based software synthesis and constraint solving to generate sets of manufacturing simulation models. In *CIRP*, pages 556–561. Elsevier, 2020. doi:10.1016/j.procir.2020.03.018.

**11** Fadil Kallat, Tristan Schäfer, and Anna Vasileva. CLS-SMT: bringing together combinatory logic synthesis and satisfiability modulo theories. In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 51–65, 2019. doi:10.4204/EPTCS.301.7.

**12** Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas W. Reps. Semantics-guided synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi:10.1145/3434311.

**13** Patrick Landwehr and Christof Löding. Tree Automata with Global Constraints for Infinite Trees. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, volume 126 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

**14** Parthasarathy Madhusudan. Synthesizing reactive programs. In Marc Bezem, editor, *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*, volume 12 of *LIPIcs*, pages 428–442. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.CSL.2011.428.

**15** Hitoshi Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2001.

**16** Hitoshi Ohsaki and Hiroyuki Seki. Languages modulo normalization. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*, volume 4720 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2007.

**17** Jakob Rehof and Pawel Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. doi:10.1007/978-3-642-21691-6_15.

**18** Jakob Rehof and Moshe Y. Vardi. Design and synthesis from components (dagstuhl seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014. doi:10.4230/DagRep.4.6.29.

**19** Tristan Schäfer, Jim A. Bergmann, Rafael G. Carballo, Jakob Rehof, and Petra Wiederkehr. A synthesis-based tool path planning approach for machining operations. In *CIRP*, pages 918–923. Elsevier, 2021. doi:10.1016/j.procir.2021.11.154.

**20** Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

**21** Sigrid Wenzel, Jana Stolipin, Jakob Rehof, and Jan Winkels. Trends in automatic composition of structures for simulation models in production and logistics. In *WSC*, pages 2190–2200. IEEE, 2019. doi:10.1109/WSC40007.2019.9004959.

# A The emptiness algorithm

In the emptiness decision algorithm, we use the following value

$$b(\mathcal{A}) = \max(\beta k + \gamma, |Q| \times |\mathcal{F}|)$$

where

- $s = s(\mathcal{A})$ is the number of *distinct* suffixes of positions $\pi, \pi'$ in an atomic constraint $\pi \neq \pi'$ in a rule of $\mathcal{A}$.
- $n = n(\mathcal{A})$ is the *maximum* number of atomic constraints in a rule of $\mathcal{A}$,
- $d = d(\mathcal{A})$ is the maximum length of $\pi$ or $\pi'$ in an atomic constraint $\pi \neq \pi'$ in a rule of $\mathcal{A}$,
- $e = \sum_{i=1}^{s} \frac{1}{i!}$,
- $\beta = (d+1)n(e|Q|2^s s! + 1)$,
- $\gamma = (2dne + 1)(d+1)n|Q|2^s s!$,
- $k = \lceil \frac{\beta + \sqrt{\beta^2 + 4\gamma}}{2} \rceil$.

The value $b(\mathcal{A})$ above is a slight improvement on [7] where a less precise bound is used. For Lemma 26 in [7], the value $k$ must satisfy $k^2 \geq h(\mathcal{A}, k)$ where

$$\begin{aligned} h(\mathcal{A}, k) &= (d+1)n(k + g(\mathcal{A}, k + 2dn)) \\ g(\mathcal{A}, k) &= (ek+1)|Q|2^s s! \end{aligned}$$

One can calculate that

$$\begin{aligned} h(\mathcal{A}, k) &= (d+1)nk + (d+1)n((e(k+2dn)+1)|Q|2^s s!) \\ &= (d+1)nk + (d+1)n(ek|Q|2^s s! + 2dne|Q|2^s s! + |Q|2^s s!) \\ &= k(d+1)n(e|Q|2^s s! + 1) + (2dne+1)(d+1)n|Q|2^s s! \\ &= \beta k + \gamma \end{aligned}$$

Hence, we need to find the smallest $k$ such that

$$k^2 - \beta k - \gamma \geq 0$$

The least integer equal or greater than the second (positive) root $\frac{\beta + \sqrt{\beta^2 + 4\gamma}}{2}$ of the quadratic equation does the job, and we obtain the $k$ listed above. According to the proofs in [7], we can then take $b(\mathcal{A}) = \max(h(\mathcal{A}, k), |Q| \times |\mathcal{F}|)$.

The pseudocode for the emptiness decision algorithm is presented in Listing 1.

**Listing 1** Emptiness decision algorithm.

```
Input: 𝒜 = (Q, Q^f, Δ).
Output: true iff L(𝒜) = ∅.

Let C be the set of suffixes of positions π, π' in atomic
constraints of transition rules in Δ.
E* ← ∅
M ← ∅
repeat
  E ← ∅
  for all r ∈ Δ do
    Let m be the arity of r (i.e. the arity of the top
    symbol in the rule).
    for all ρ_1, ..., ρ_m ∈ E* s.t. r(ρ_1, ..., ρ_m) is a run do
      ρ ← r(ρ_1, ..., ρ_m)
      if ρ ∈ M then
        continue
      endif
      M ← M ∪ {ρ}
      v ← true
      for all p ∈ Pos(ρ) \ C s.t. |p| ≤ d + 1 do
        for all ρ'_1, ..., ρ'_b ∈ E* s.t. all ρ'_i(ϵ) have the
                same target state as ρ(p)
        do
          if ρ|_p ≫ ρ'_b ≫ ... ≫ ρ'_1 and
             for all 1 ≤ j ≤ b, ρ[ρ'_j]_p does not contain any
             equality close to p
          then
            v ← false
          endif
        done
      done
      if v then
        E ← E ∪ {ρ}
      endif
    done
  done
  E* ← E* ∪ E
until E = ∅
if E* contains an accepting run then
  return false
else
  return true
endif
```

# On Lookaheads in Regular Expressions with Backreferences

**Nariyoshi Chida**
NTT Corporation, Tokyo, Japan
Waseda University, Tokyo, Japan

**Tachio Terauchi**
Waseda University, Tokyo, Japan

## Abstract

Many modern regular expression engines employ various extensions to give more expressive support for real-world usages. Among the major extensions employed by many of the modern regular expression engines are *backreferences* and *lookaheads*. A question of interest about these extended regular expressions is their expressive power. Previous works have shown that (i) the extension by lookaheads does not enhance the expressive power, i.e., the expressive power of regular expressions with lookaheads is still regular, and that (ii) the extension by backreferences enhances the expressive power, i.e., the expressive power of regular expressions with backreferences (abbreviated as *rewb*) is no longer regular. This raises the following natural question: Does the extension of regular expressions with backreferences by lookaheads enhance the expressive power of regular expressions with backreferences? This paper answers the question positively by proving that adding either positive lookaheads or negative lookaheads increases the expressive power of rewb (the former abbreviated as $rewbl_p$ and the latter as $rewbl_n$). A consequence of our result is that neither the class of finite state automata nor that of memory automata (MFA) of Schmid [14] (which corresponds to regular expressions with backreferenes but without lookaheads) corresponds to $\text{rewbl}_p$ or $\text{rewbl}_n$. To fill the void, as a first step toward building such automata, we propose a new class of automata called *memory automata with positive lookaheads* (PLMFA) that corresponds to $\text{rewbl}_p$. The key idea of PLMFA is to extend MFA with a new kind of memories, called *positive-lookahead memory*, that is used to simulate the backtracking behavior of positive lookaheads. Interestingly, our positive-lookahead memories are almost perfectly *symmetric* to the capturing-group memories of MFA. Therefore, our PLMFA can be seen as a natural extension of MFA that can be obtained independently of its original intended purpose of simulating $\text{rewbl}_p$.

**2012 ACM Subject Classification** Theory of computation → Formal languages and automata theory

**Keywords and phrases** Regular expressions, Lookaheads, Backreferences, Memory automata

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.15

**Funding** This work was supported by JSPS KAKENHI Grant Numbers 17H01720, 18K19787, 20H04162, 20K20625, and 22H03570.

## 1 Introduction

Regular expressions, introduced by Kleene [9], and the extensions employed by many of the modern regular expression engines are widely studied in formal language theory. Among the major extensions are *backreferences* and *lookaheads*. Previous works on formal language theory have studied the two features mostly in isolation. Morihata [12] and Berglund et al. [3] showed that extending regular expressions by lookaheads does not enhance their expressive power. Their proofs are by a translation to boolean finite automata [4] whose expressive power is regular. The formal study of regular expressions with backreferences (rewb) dates back to the seminal work by Aho [1]. More recently, a formal semantics and a pumping lemma were given by Câmpeanu et al. [5], and Berglund and van der Merwe [2] showed that different variants of backreference semantics give rise to differences in expressive powers. Schmid [14] proposed *memory automata* (MFA) and showed that the expressive power of the automata is equivalent to that of rewb.

© Nariyoshi Chida and Tachio Terauchi;
licensed under Creative Commons License CC-BY 4.0

In this paper, we initiate a formal study of *regular expression with backreferences and lookaheads* (*rewbl* for short). We call the fragment containing only positive (resp. negative) lookaheads *rewbl*$_p$ (resp. *rewbl*$_n$). We show that both rewbl$_p$ and rewbl$_n$ are more expressive than rewb, and also prove some language-theoretic properties of rewbl. One consequence of the results is the undecidability of a problem tackled in a recent work [11].

Another consequence of our results is that neither the class of finite state automata nor that of memory automata (MFA) of Schmid [14] (which corresponds to regular expressions with backreferences but without lookaheads) corresponds to rewbl$_p$ or rewbl$_n$. As remarked above, prior works [3, 12] have applied translation to boolean finite automata [4] (or alternating finite automata [7]) to build automata equivalent to regular expressions with lookaheads. They simulate lookaheads by executing multiple runs simultaneously without backtracking. Unfortunately, the interaction of lookaheads with backreferences prevents us from applying the approaches to rewbl. Namely, rewbl permits *cross-lookahead backreferences* whereby a string captured outside of a lookahead is referred from inside of the lookahead, or vice versa (only the former is allowed for negative lookaheads whereas both are allowed for positive lookaheads). Such cross-lookahead backreferences intrinsically require backtracking. In our work, as a first step toward building automata equivalent to rewbl, we introduce a new class of automata called *memory automata with positive lookaheads* (PLMFAs). We prove that PLMFAs are equivalent to rewbl$_p$ in expressive power. A key component of PLMFAs is a new kind of memories, called a *positive-lookahead memory*, that is used to simulate the backtracking behavior of positive lookaheads. Interestingly, our positive-lookahead memories are almost perfectly *symmetric* to the capturing-group memories of MFA. Therefore, our PLMFA can be seen as a natural extension of MFA that can be obtained independently of its original intended purpose of simulating rewbl$_p$.

In summary, this paper makes the following contributions:

- We show that the extension of rewb by either positive or negative lookaheads enhances the expressive power. Additionally, we prove some language-theoretic properties of rewbl. (Sec. 3)
- We introduce memory automata with positive lookaheads (PLMFAs), a new class of automata that we prove to be equivalent in expressive power to rewbl$_p$. A key component of PLMFAs is a new kind of memories called positive-lookahead memory, which is almost perfectly symmetric to capturing-group memory of MFA. (Sec. 4)

We believe that our work leads to interesting future developments in both theoretical and practical fronts: interesting practically because backreferences and lookaheads are practically motivated by real-world needs, and interesting theoretically because, as we shall show, rewbl does not appear to correspond to any known formal language classes.

## 2 Preliminaries

In this section, we introduce the preliminary notations (Sec. 2.1) and present the syntax and the semantics of rewbl (Sec. 2.2).

### 2.1 Notation

We write $\mathbb{N}$ for the set of natural numbers and $[i]$ for the set $\{1, 2, \dots, i\}$ where $i \in \mathbb{N}$. For a sequence $l$, we write $|l|$ for its length, $l[i]$ (for $1 \le i \le |l|$) for its $i$th element, $l[i..j]$ for the sub-sequence from the $i$th element to the $j$th element (for $1 \le i \le j \le |l|$). We write $l_1 :: l_2$ for the concatenation of $l_1$ and $l_2$. We abbreviate it as $l_1 l_2$ if clear from the context. We write $v \in l$ to denote that $l$ contains $v$. We write $\Sigma$ for a finite alphabet; $a, b \in \Sigma$ for a

| $r$ | $::=$ | $a$ | character | $\mid$ | $r^*$ | repetition |
|---|---|---|---|---|---|---|
| | $\mid$ | $\emptyset$ | empty set | $\mid$ | $({}_i r)_i$ | capturing group |
| | $\mid$ | $\epsilon$ | empty string | $\mid$ | $\backslash i$ | backreference |
| | $\mid$ | $rr$ | concatenation | $\mid$ | $(?{=}r)$ | pos-lookahead |
| | $\mid$ | $r\mid r$ | union | $\mid$ | $(?!r)$ | neg-lookahead |

**Figure 1** The syntax of rewbl expressions.

character; $x, y \in \Sigma^*$ for a sequence of characters (i.e., *string*); $\epsilon$ for the empty string; $\Sigma_\epsilon$ for $\Sigma \cup \{\epsilon\}$; In what follows, we fix a finite alphabet $\Sigma$. For $1 \le i < j \le |x|$, we define $x[i..j)$ to be $x[i...j-1]$. For $x, y \in \Sigma^*$, we define $x\backslash y$ to be the left quotient of $x$ divided by $y$, i.e., $v$ where $yv = x$. Dually, the right quotient of $x$ divided $y$, $x/y$, is $v$ where $vy = x$. $S \subset U$ denotes that $S$ is a proper subset of $U$, i.e., $S \subseteq U \wedge S \neq U$. For a partial map $f$, we write $dom(f)$ for the domain of $f$. $\mathcal{P}(S)$ denotes that the power set of a set $S$. For $f$ a (partial) function, $f[\alpha \mapsto \beta]$ denotes the (partial) function that maps $\alpha$ to $\beta$ and behaves as $f$ for all other arguments. We write $f(\alpha) = \bot$ if $f$ is undefined at $\alpha$.

## 2.2 Regular Expressions with Backreferences and Lookaheads

The syntax of *regular expressions with backreferences and lookaheads* (*rewbl*) is given by Fig. 1. The semantics of the pure regular expression constructs (i.e., the first six constructs of Fig. 1) is standard. We write $\cdot$ for the rewbl that matches any character (i.e., $a_1|\dots|a_n$ where $\Sigma = \{a_1, \dots, a_n\}$). The precedence order of the operators is as follows: Kleene-*, concatenation, and union. The left has a higher precedence. For example, the expression $a^*|bc^*$ means $((a^*)|(b(c^*)))$ due to the priority.

The remaining constructs, i.e., capturing groups, backreferences, and lookaheads, are the extensions considered in this paper. In conformance with the nomenclature from the literature [10], we call the fragment of rewbl without lookaheads *rewb*. We call the fragment of rewbl without negative (resp. positive) lookaheads $\mathit{rewbl}_p$ (resp. $\mathit{rewbl}_n$). In what follows, we explain the semantics of the extended features informally in terms of the standard backtracking-based matching algorithm which attempts to match the given regular expression with the given string and backtracks when the attempt fails. A *capturing group* $({}_i r)_i$ (or $(r)_i$ if no ambiguity arises) attempts to match $r$, and if successful, stores the matched substring in the storage identified by the index $i$. Otherwise, the match fails and the algorithm backtracks. A *backreference* $\backslash i$ refers to the substring matched to the corresponding capturing group $({}_i r)_i$, and attempts to match the same substring if the capture had succeeded. If the capture had not succeeded, i.e., is an *unassigned backreference*, or the matching against the captured substring fails, then the algorithm backtracks. Capturing groups in practice often do not have explicit indexes, but we write them here for readability. A *positive* (resp. *negative*) *lookahead* $(?{=}r)$ (resp. $(?!r)$) attempts to match $r$ without any character consumption, proceeds if the match succeeds (resp. fails), and backtracks otherwise.

More formally, the semantics is defined by the *matching relation* $\rightsquigarrow$ that models the behavior of backtracking matching algorithms. The full rules for deriving the matching relation $\rightsquigarrow$ is shown in Fig. 2. The semantics is same as the one defined in our recent work [8] except for specializing the set-of-characters rules to the rules for a single character and the empty set. We define $\mathit{ite}(\mathit{true}, A, B) = A$ and $\mathit{ite}(\mathit{false}, A, B) = B$.

A matching relation is of the form $(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}$ where $p$ is a position on the string $w$ such that $1 \le p \le |w| + 1$, $\Lambda$, called an *environment*, is a function that maps each capturing group index to a string captured by the corresponding capturing group, and $\mathcal{N}$ is a set of

$$\frac{p \leq |w| \qquad w[p] = a}{(a, w, p, \Lambda) \rightsquigarrow \{(p+1, \Lambda)\}} \text{ (Character)}$$

$$\frac{p > |w| \vee w[p] \neq a}{(a, w, p, \Lambda) \rightsquigarrow \emptyset} \text{ (Character Failure)}$$

$$\frac{}{(\emptyset, w, p, \Lambda) \rightsquigarrow \emptyset} \text{ (Empty Set)}$$

$$\frac{}{(\epsilon, w, p, \Lambda) \rightsquigarrow \{(p, \Lambda)\}} \text{ (Empty String)}$$

$$\frac{(r_1, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad \forall (p_i, \Lambda_i) \in \mathcal{N},\ (r_2, w, p_i, \Lambda_i) \rightsquigarrow \mathcal{N}_i}{(r_1 r_2, w, p, \Lambda) \rightsquigarrow \bigcup_{0 \leq i < |\mathcal{N}|} \mathcal{N}_i} \text{ (Concatenation)}$$

$$\frac{(r_1, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad (r_2, w, p, \Lambda) \rightsquigarrow \mathcal{N}'}{(r_1 | r_2, w, p, \Lambda) \rightsquigarrow \mathcal{N} \cup \mathcal{N}'} \text{ (Union)}$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N} \quad \forall (p_i, \Lambda_i) \in (\mathcal{N} \backslash \{(p, \Lambda)\}),\ (r^*, w, p_i, \Lambda_i) \rightsquigarrow \mathcal{N}_i}{(r^*, w, p, \Lambda) \rightsquigarrow \{(p, \Lambda)\} \cup \bigcup_{0 \leq i < |(\mathcal{N} \backslash \{(p, \Lambda)\})|} \mathcal{N}_i} \text{ (Repetition)}$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}}{((r)_j, w, p, \Lambda) \rightsquigarrow \{(p_i, \Lambda_i[j \mapsto w[p..p_i)]) \mid (p_i, \Lambda_i) \in \mathcal{N}\}} \text{ (Capturing Group)}$$

$$\frac{\Lambda(i) \neq \bot \qquad (\Lambda(i), w, p, \Lambda) \rightsquigarrow \mathcal{N}}{(\backslash i, w, p, \Lambda) \rightsquigarrow \mathcal{N}} \text{ (Backreference)}$$

$$\frac{\Lambda(i) = \bot}{(\backslash i, w, p, \Lambda) \rightsquigarrow \emptyset} \text{ (Backreference Failure)}$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}}{((?{=}r), w, p, \Lambda) \rightsquigarrow \{(p, \Lambda') \mid (_, \Lambda') \in \mathcal{N}\}} \text{ (Positive Lookahead)}$$

$$\frac{(r, w, p, \Lambda) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset, \emptyset, \{(p, \Lambda)\})}{((?!r), w, p, \Lambda) \rightsquigarrow \mathcal{N}'} \text{ (Negative Lookahead)}$$

**Figure 2** Rules of the matching relation $\rightsquigarrow$.

matching results. A *matching result* is a pair of a position and an environment. Roughly, $(r, w, p, \Lambda) \rightsquigarrow \mathcal{N}$ is read: a rewbl expression $r$ tries to match the string $w$ from the position $p$, with the environment $\Lambda$ and, if $(p', \Lambda') \in \mathcal{N}$, $r$ consumed $p' - p$ characters and updated the environment to $\Lambda'$. Additionally, if $\mathcal{N} = \emptyset$, it means that the matching failed.

In the two rules for a character, the rewbl $a$ tries to match the string $w$ at the position $p$ with the function capturing $\Lambda$. If the $p$th character $w[p]$ is $a$, then the matching succeeds returning the matching result $(p+1, \Lambda)$ (Character). Otherwise, the character $w[p]$ does not match or the position is at the end of the string, and $\emptyset$ is returned as the matching result indicating the match failure (Character Failure).

The rules (Empty Set), (Empty String), (Concatenation), (Union) and (Repetition) are self explanatory. Note that we avoid self looping in (Repetition) by not repeating the match from the same position.

In the rule (Capturing Group), we first get the matching result $\mathcal{N}$ from matching $w$ against $r$ at the current position $p$. And for each matching result $(p_i, \Lambda_i) \in \mathcal{N}$ (if any), we record the matched substring $w[p..p_i)$ in the corresponding environment $\Lambda_i$ at the index $i$. The rule (Backreference) looks up the captured substring and tries to match it with the input at the current position. The match fails if the corresponding capture has failed as stipulated by the rule (Backreference Failure).

In the rule (Positive Lookahead), the expression $r$ is matched against the given string $w$ at the current position $p$ to obtain the matching results $\mathcal{N}$. Then, for every match result $(p', \Lambda') \in \mathcal{N}$ (if any), we reset the position from $p'$ to $p$. This models the behavior of lookaheads which does not consume the string. The rule (Negative Lookahead) is similar, except that we reset and proceed when there is no match. Note that captures made inside of a negative lookahead cannot be referred outside of the lookahead, which agrees with the behavior of regular expression engines in practice.

▶ **Definition 1** (Language)**.** The *language* of a rewbl $r$ is defined as $L(r) = \{w \mid (r, w, 1, \emptyset) \rightsquigarrow \mathcal{N} \wedge \exists \Lambda.(|w|+1, \Lambda) \in \mathcal{N}\}$.

Recall that one subtle aspect of rewbl is that backreferences can cross lookahead boundaries (cf. Sec. 1). We next show some examples of cross-lookahead backreferences.

▶ **Example 2.** Consider the expression $(_1 .^* \mathtt{z})_1 (?{=}\backslash 1) .^*$. Its language is $\{x\mathtt{z}x\mathtt{z}y \mid x, y \in \Sigma^*\}$. For example, when the input string is `azazbc`, the expression captures the prefix `az` and refers it from the inside of the positive lookahead `(?=\1)`.

▶ **Example 3.** Consider the expression $(_1\cdot)_1$`(?!\1)`$\cdot^*$. The language is $\{a \mid a \in \Sigma\} \cup \{abx \mid a, b \in \Sigma \wedge a \neq b \wedge x \in \Sigma^*\}$.

▶ **Example 4.** Consider the expression `(?=`$(_1\cdot^*)_1$`z)`$\cdot$`\1`. The language is $\{\mathtt{z}x \mid x \in \{\mathtt{z}\}^*\}$.

Example 2 (resp. 3) shows an example where a string captured outside of a positive (resp. negative) lookahead is backreferenced in the lookahead. Example 4 shows an example where a string captured inside of a positive lookahead is backreferenced from outside of the lookahead.

### 2.2.1 Conventions on Syntax and Semantics

We review the conventions regarding capturing groups and unassigned references. The conventions are proposed in prior works on rewb, and as shown by [2], they affect the expressive power of rewb. Here, we simply present the conventions and refer interested readers to [2] for the expressive power differences.

There are two convensions regarding capturing groups: *no label repetitions (NLR)* and *may repeat labels (MRL)*. NLR requires the indexes of capturing groups to be distinct, whereas MRL imposes no such restrictions. For example, $(\cdot^*)_1$`\1` satisfies NLR, but $((\cdot^*)_1|(\cdot^*)_1)$`\1` does not because the capturing group with index 1 appears twice. NLR is assumed in the prior works by Câmpeanu et al. [5] and Carle and Narendran [6] on the expressive power of rewb.

There are two conventions regarding unassigned references: the $\epsilon$ *semantics* and the $\emptyset$ *semantics*. The $\epsilon$ (resp. $\emptyset$) semantics defines that unassigned references are handled as an empty string $\epsilon$ (resp. a failure $\emptyset$). For example, for $r = \mathtt{a}\backslash\mathtt{1}$, $L(r) = \{\mathtt{a}\}$ with the $\epsilon$ semantics but $L(r) = \emptyset$ with the $\emptyset$ semantics. Additionally, prior works have proposed a condition called *no unassigned reference (NUR)*. The NUR condition does not allow unassigned references in expressions, i.e., all expressions with unassigned references are to be excluded (see below for the formal definition). For example, $r = \mathtt{a}\backslash\mathtt{1}$ does not satisfy the NUR condition because `\1` is an unassigned references. Note that the $\epsilon$ semantics and the $\emptyset$ semantics coincide under the NUR condition because there would be no unassigned references. The condition is also assumed in [5, 6] ([6] incorrectly remarks that [5] does not assume the condition).

In the rest of this section, we give a formal definition of the NUR condition that we shall also use later in our proofs. We note that prior works that proposed the condition did not provide a formal definition of it [5, 6]. First, we define the function *Capture* from rewbls to the set of capturing group indexes that can be referred from their continuations:

$$Capture(r) = \begin{cases} \emptyset & (\text{if } r = a, \emptyset, \epsilon, r_1^*, \backslash i, \text{ or } (?!r_1)) \\ Capture(r_1) \cup Capture(r_2) & (\text{if } r = r_1 r_2) \\ Capture(r_1) \cap Capture(r_2) & (\text{if } r = r_1|r_2) \\ Capture(r_1) \cup \{i\} & (\text{if } r = (_i r_1)_i) \\ Capture(r_1) & (\text{if } r = (?{=}r_1)) \end{cases}$$

With this, we can define the predicate NUR($S$,$r$) that says that $r$ satisfies NUR condition if it occurs in a context where the capturing group indexes in $S$ can be referred:

$$\mathrm{NUR}(S, r) = \begin{cases} \mathit{true} & (\text{if } r = a, \emptyset, \text{ or } \epsilon) \\ \mathrm{NUR}(S, r_1) \wedge \mathrm{NUR}(S \cup Capture(r_1), r_2) & (\text{if } r = r_1 r_2) \\ \mathrm{NUR}(S, r_1) \wedge \mathrm{NUR}(S, r_2) & (\text{if } r = r_1|r_2) \\ \mathrm{NUR}(S, r_1) & (\text{if } r = r_1^*, (_i r_1)_i, (?{=}r_1), \text{ or } (?!r_1)) \\ i \in S & (\text{if } r = \backslash i) \end{cases}$$

Then, $r$ can be said to satisfy the NUR condition iff $\mathrm{NUR}(\emptyset, r) = \mathit{true}$. For example, the expression $r = (_1\mathtt{a})_1\backslash 1$ satisfies the NUR condition because $\mathrm{NUR}(\emptyset, r) = \mathrm{NUR}(\emptyset, (_1\mathtt{a})_1) \land \mathrm{NUR}(\emptyset \cup \mathit{Capture}((_1\mathtt{a})_1), \backslash 1) = \mathrm{NUR}(\emptyset, \mathtt{a}) \land \mathrm{NUR}(\{1\}, \backslash 1) = \mathit{true}$. As another example, $r = (_1\mathtt{a}\backslash 1)_1$ does not satisfy the NUR condition because $\mathrm{NUR}(\emptyset, r) = \mathrm{NUR}(\emptyset, \mathtt{a}\backslash 1) = \mathrm{NUR}(\emptyset, \mathtt{a}) \land \mathrm{NUR}(\emptyset \cup \mathit{Capture}(\mathtt{a}), \backslash 1) = \mathit{false}$. In what follows, unless explicitly stated otherwise, we assume that a rewbl that satisfies NLR and NUR.

## 3 Language Properties of Rewbl

In this section, we prove some salient language properties of rewbl. Importantly, we show that both rewbl$_p$ and rewbl$_n$ is strictly more expressive than rewb, thus showing that the extension by either posistive or negative lookaheads changes the expressive power of rewb. In the following, we denote by $\mathbb{L}_B$, $\mathbb{L}_{BL}$, $\mathbb{L}_{BL_n}$, and $\mathbb{L}_{BL_p}$ the class of languages matched by rewb, rewbl, rewbl$_n$, and rewbl$_p$, respectively.

Our first result states that $\mathbb{L}_{BL_n}$ is closed under union, intersection, and complement.

▶ **Theorem 5.** *$\mathbb{L}_{BL_n}$ is closed under union, intersection, and complement.*

**Proof.** Suppose we have rewbl expressions $r_1$ and $r_2$. Then, rewbl expressions that accept the union of $L(r_1)$ and $L(r_2)$, the intersection of $L(r_1)$ and $L(r_2)$, and the complement of $L(r_1)$ can be constructed respectively as follows.

- **Union:** $L(r_1) \cup L(r_2) = L(r_1|r_2)$;
- **Intersection:** $L(r_1) \cap L(r_2) = L((?!(?!r_1(?!\cdot)))r_2)$; and
- **Complement:** $L(r_1)^c \triangleq \Sigma^* \backslash L(r_1) = L((?!r_1(?!\cdot))\cdot^*)$.

In **Intersection** and **Complement**, a subtle point is that a negative lookahead $(?!r)$ accepts a string even if the expression $r$ rejects only a prefix of the string. For example, $L((?!(?!r_1))r_2)$ is the set of strings in $L(r_2)$ that have a prefix that belongs to $L(r_1)$, rather than the intersection of $L(r_1)$ and $L(r_2)$. To force whole matching, the negative lookahead $(?!\cdot)$ is appended. ◀

Of course, we could alternatively show **Intersection** from **Union** and **Complement** by applying De Morgan's laws: $L(r_1) \cap L(r_2) = (L(r_1)^c \cup L(r_2)^c)^c$. The above proof gives a direct construction which shows that the intersection can be obtained by a short rewbl$_n$ expression.

We next show that $\mathbb{L}_{BL}$ is also closed under union, intersection, and complement.

▶ **Theorem 6.** *$\mathbb{L}_{BL}$ is closed under union, intersection, and complement.*

**Proof.** The proof is the same as Theorem 5. Or, for **Intersection**, an even shorter proof is possible: $L(r_1) \cap L(r_2) = L((?{=}r_1(?!\cdot))r_2)$. ◀

A consequence of Theorem 5 is that rewbl and rewbl$_n$ are more expressive than rewb.

▶ **Corollary 7.** $\mathbb{L}_B \subset \mathbb{L}_{BL_n} \subseteq \mathbb{L}_{BL}$.

**Proof.** Immediate from Theorem 5 and Lemma 3 of [6] which showed that rewb is not closed under intersection. ◀

We show that adding just positive lookaheads also increases the expressive power of rewb.

▶ **Theorem 8.** $\mathbb{L}_B \subset \mathbb{L}_{BL_p}$.

**Table 1** Summary of closure properties. The rows highlighted in gray present our new results. ? indicates that the problem is open.

| | Closure under | | | | |
|---|---|---|---|---|---|
| | $\cup$ | $\cap$ | Complement | Concatenation | Kleene-* |
| Regular | Yes | Yes | Yes | Yes | Yes |
| $\mathbb{L}_B$ | Yes | No | No | Yes | Yes |
| $\mathbb{L}_{BL_p}$ | Yes | ? | ? | Yes | Yes |
| $\mathbb{L}_{BL_n}$ | Yes | Yes | Yes | Yes | Yes |
| $\mathbb{L}_{BL}$ | Yes | Yes | Yes | Yes | Yes |

**Proof.** From [6], the language $S = \{a^i b a^{i+1} b a^k \mid k = i(i+1)k', k' > 0, \text{ and } i > 0\}$ is not in $\mathbb{L}_B$. Let $S' = \{a^i b a^{i+1} b a^k c \mid k = i(i+1)k', k' > 0, \text{ and } i > 0\}$. We can prove that $S'$ is also not in $\mathbb{L}_B$ in a manner similar to the proof that $S$ is not in $\mathbb{L}_B$ [6].

Now, $S'$ is the intersection of $L(r_1c)$ and $L(r_2c)$ where $r_1$ and $r_2$ are $(aa^*)_1 b\backslash 1ab\backslash 1\backslash 1^*$ and $(aa^*)_1 b(\backslash 1a)_2 b\backslash 2\backslash 2^*$, respectively. Then, the intersection of the languages of $r_1c$ and $r_2c$ is $L((?{=}r_1c)r_2c) = S' \in \mathbb{L}_{BL_p}$. ◀

A summary of the results of closure properties can be found in Table 1. Additionally, the diagram below summarizes the results of the hierarchy of the language classes. Here, $\mathbb{L}_L$ denotes the class of languages matched by regular expressions with (both positive and negative) lookaheads, which is known to be equivalent in expressive power to the set of regular languages [3, 12].

$$\text{Regular} = \mathbb{L}_L \subset \mathbb{L}_B \;\; \begin{matrix} \subset \; \mathbb{L}_{BL_p} \; \subseteq \\ \subset \; \mathbb{L}_{BL_n} \; \subseteq \end{matrix} \;\; \mathbb{L}_{BL}$$

From Theorems 5 and 6, we also obtain the following result.

▶ **Theorem 9.** *The emptiness problems of rewbl and rewbl$_n$ are undecidable.*

**Proof.** Suppose for contradiction that the emptiness problem of rewbl is decidable. By Theorem 6, we know that the language of rewbl is closed under intersection. Therefore, the emptiness problem of intersection of two rewb expressions is also decidable. However, this contradicts a result from [6] which states that the latter problem is undecidable. The proof for rewbl$_n$ is similar by using Theorem 5. ◀

A recent work [11] has proposed a method for symbolically executing programs containing rewbl. Their method generates and tries to solve constraints of the form $x \in L(r)$ where $r$ is a rewbl expression and $x$ is a variable for which the method tries to find an assignment that satisfies the constraint. Theorem 9 implies that their constraint solving problem is undecidable.

▶ **Corollary 10.** *The constraint solving problem of [11] is undecidable.*

▶ Remark 11. The constructions used to show Cor. 7 and Theorem 8 do not contain backreferences that cross lookahead boundaries (recall the discussion from Sec. 1 and Sec. 2.2). Thus, our results show that lookaheads enhance the expressive power of rewb even without cross-lookahead backreferences. We leave for future work to investigate whether there are expressive power changes from allowing or disallowing cross-lookahead backreferences.

## 3.1 Restricted Label Repetitions

As pointed out in [2], allowing rewbs to repeat labels of backreferences affects their expressive powers. In this sub-section, we introduce new conditions called *restricted may repeat labels* (RMRL) and *no self-capturing reference* (NSR). RMRL allows repeating the labels but only in a restricted way. NSR enforces that there is no reference that is nested by the capturing group of the same index. We use RMRL and NSR as an intermediary in the construction of automata equivalent in expressive power to rewbl$_p$ (with NUR and NLR) in the next section. But the new conditions may also be of independent interest.

Informally, RMRL requires the capturing group that is referred to by any reference is uniquely determined. For example, $(_1a)_1(_1b)_1\backslash 1$ satisfies RMRL because the reference $\backslash 1$ refers to the capturing group $(_1b)_1$ while $((_1a)_1|(_1b)_1)\backslash 1$ does not satisfy RMRL because the reference $\backslash 1$ can refer to the capturing groups $(_1a)_1$ and $(_1b)_1$. To represent the repetitions of indexes, we use a *multiset*. A multiset, denoted by $\{\!\!\{ \cdots \}\!\!\}$, is a collection of elements with repetitions. For example, $\{\!\!\{a, a, b\}\!\!\}$ is a multiset that has two $a$'s and one $b$. We use the notation for sets as that of multisets, e.g., we use $\cup$ for the union of multisets, e.g., $\{\!\!\{a\}\!\!\} \cup \{\!\!\{a, b\}\!\!\} = \{\!\!\{a, a, b\}\!\!\}$, and $|\cdot|$ for the number of elements, e.g., $|\{\!\!\{a, a, b\}\!\!\}| = 3$. Formally, we define RMRL by first defining *NumCaps* that takes a multiset $\mathbb{S}$ and a rewbl expression $r$ and returns a multiset that represents the number of ways to capture for each index.

$$
NumCaps(\mathbb{S}, r) = \begin{cases} \mathbb{S} & (\text{if } r = a, \emptyset, \epsilon, \backslash i, \text{ or } (?!r_1)) \\ NumCaps(NumCaps(\mathbb{S}, r_1), r_2) & (\text{if } r = r_1 r_2) \\ NumCaps(\mathbb{S}, r_1) \cup NumCaps(\mathbb{S}, r_2) & (\text{if } r = r_1 | r_2) \\ \{\!\!\{ j \mid j \in NumCaps(\mathbb{S}, r_1) \wedge j \neq i \}\!\!\} \cup \{\!\!\{ i \}\!\!\} & (\text{if } r = (_i r_1)_i) \\ NumCaps(\mathbb{S}, r_1) & (\text{if } r = (?{=}r_1) \text{ or } r_1^*) \end{cases}
$$

With this, we define RMRL($\mathbb{S}$,$r$) that says that $r$ satisfies the RMRL condition if it occurs in a context where the capturing group indexes in $\mathbb{S}$ can be referred:

$$
\text{RMRL}(\mathbb{S}, r) = \begin{cases} \mathit{true} & (\text{if } r{=}a, \emptyset, \text{ or } \epsilon) \\ \text{RMRL}(\mathbb{S}, r_1) \wedge \text{RMRL}(NumCaps(\mathbb{S}, r_1), r_2) & (\text{if } r{=}r_1 r_2) \\ \text{RMRL}(\mathbb{S}, r_1) \wedge \text{RMRL}(\mathbb{S}, r_2) & (\text{if } r{=}r_1|r_2) \\ \text{RMRL}(\mathbb{S}, r_1) & (\text{if } r{=}r_1^*, (_i r_1)_i, (?{=}r_1), \text{ or } (?!r_1)) \\ |\{\!\!\{ i \mid i \in \mathbb{S} \}\!\!\}| \leq 1 & (\text{if } r{=}\backslash i) \end{cases}
$$

We say that a rewbl expression $r$ satisfies RMRL iff $\text{RMRL}(\emptyset, r) = \mathit{true}$.

Next, we explain NSR. NSR requires that for every reference $\backslash i$, the reference is not nested by the capturing group whose index is $i$. For example, $(_1a\backslash 1)_1(_2\backslash 1)_2$ does not satisfy NSR because $\backslash 1$ appears in its capturing group. Formally, we define NSR as follows.

$$
\text{NSR}(S, r) = \begin{cases} \mathit{true} & (\text{if } r = a, \emptyset, \text{ or } \epsilon) \\ \text{NSR}(S, r_1) \wedge \text{NSR}(S, r_2) & (\text{if } r = r_1 r_2 \text{ or } r_1|r_2) \\ \text{NSR}(S, r_1) & (\text{if } r = r_1^*, (?{=}r_1), \text{ or } (?!r_1)) \\ \text{NSR}(S \cup \{i\}, r_1) & (\text{if } r = (_i r_1)_i) \\ i \notin S & (\text{if } r = \backslash i) \end{cases}
$$

We say that a rewbl expression $r$ satisfies NSR iff $\text{NSR}(\emptyset, r) = \mathit{true}$. We show that RMRL $\wedge$ NSR is equivalent to NLR in expressive powers.

▶ **Lemma 12.** *(1) For any NLR $r$ there exists a NSR and RMRL $r'$ such that $L(r) = L(r')$, and (2) for any NSR and RMRL $r$ there exists a NLR rewbl $r'$ such that $L(r) = L(r')$.*

**Proof.** (1) is immediate since NLR implies both RMRL and NSR (under the NUR assumption). To see (2), if $r$ satisfies RMRL and NSR, then capturing groups that are referred are uniquely determined and are closed when they are referred. Thus, we can construct $r'$ by replacing indexes of reference $i$ in $r$ and the capturing group referred to $i$ with unique indexes and removing all unreferred capturing groups. ◀

# 4 Memory Automata with Positive Lookaheads

This section present PLMFA, a new class of automata that we prove to be equivalent to rewbl$_p$. PLMFA is obtained by extending MFA of Schmid [14] that is equivalent to rewb. The key extension is the addition of a new kind of memories called *positive-lookahead memories*. Roughly, a PLMFA is a non-deterministic finite state automata augmented with a list of capturing-group memories and a list of positive-lookahead memories. The former also exists in MFA and stores strings captured by capturing groups to simulate the behavior of backreferences. The latter stores strings matched by positive lookaheads and is used to simulate the behavior of positive lookaheads.

## 4.1 Formal Definition

A *memory* is a tuple $(x, \boldsymbol{s})$ of a string $x \in \Sigma^*$ and a *status* $\boldsymbol{s}$. A status is either *open* ($\boldsymbol{O}$) or *close* ($\boldsymbol{C}$). The statuses are changed by *memory instructions* (*instructions* for short) $\Theta = \{\mathsf{o}, \mathsf{c}, \diamond\}$ as follows: $\boldsymbol{s} \oplus \mathsf{o} = \boldsymbol{O}$, $\boldsymbol{s} \oplus \mathsf{c} = \boldsymbol{C}$, and $\boldsymbol{s} \oplus \diamond = \boldsymbol{s}$. Roughly, $\boldsymbol{O}$ means that the string in the memory is modified by appending consumed strings, while $\boldsymbol{C}$ means that the string in the memory is unmodified. Changing the status from $\boldsymbol{C}$ to $\boldsymbol{O}$ (resp. from $\boldsymbol{O}$ to $\boldsymbol{C}$) representing to an entering (resp. exiting) a capturing group if the memory is a capturing-group memory and otherwise (i.e., if positive-lookahead memory) a positive lookahead. At computation steps corresponding to backreferences, the strings in capturing group memories are used to left-divide the input string and appended to strings stored in any open memories. Symmetrically, when the strings in positively lookahead memories are used, they are prepended to the input string and used to right-divide strings stored in any open memories. A positive lookahead memory is used when it gets closed. For a memory $t = (x, \boldsymbol{s})$, we write $t.word$ for $x$ and $t.status$ for $\boldsymbol{s}$. We define PLMFAs as follows.

▶ **Definition 13** (PLMFA). For $(k_c, k_p) \in \mathbb{N}^2$, a $(k_c, k_p)$*-memory automaton with positive lookaheads*, PLMFA$(k_c, k_p)$, is a tuple $(Q, \delta, q_0, F)$ such that

1. $Q$ is a finite set of *states*,
2. $\delta : Q \times (\Sigma_\epsilon \cup [k_c]) \to \mathcal{P}(Q \times \Theta^{k_c} \times \Theta^{k_p})$ is the *transition function*,
3. $q_0 \in Q$ is the *initial state*, and
4. $F \subseteq Q$ is the set of *accepting states*.

Here, $k_c$ and $k_p$ represent the number of capturing-group memories and positive-lookahead memories, respectively. Next, we define *configurations* of PLMFAs.

▶ **Definition 14** (Configuration). A *configuration* of a PLMFA $M$ is a tuple $(q, w, \sigma_c, \sigma_p)$ where $q$ is a state of $M$, $w$ is an input string, and $\sigma_c$ (resp. $\sigma_p$) is a list of memories that represents a list of capturing-group (resp. positive-lookahead) memories.

▶ **Definition 15** (Computation step). For a PLMFA$(k_c, k_p)$ $M$ and $\ell \in \Sigma_\epsilon \cup [k_c]$, a *step of computation* of $M$ is a binary relation on configurations $\xrightarrow[M]{\ell}$ (or $\xrightarrow{\ell}$ or $\to$ if irrelevant), defined as follows: $(q, w, \sigma_c, \sigma_p) \xrightarrow[M]{\ell} (q', w', \sigma'_c, \sigma'_p)$ iff there is a transition $\delta(q, \ell) \ni (q', ir_c, ir_p)$ satisfying the following conditions.

**(1)** If $\ell \in \Sigma_\epsilon$, $v = \ell$ and otherwise (i.e., if $\ell \in [k_c]$) if $\sigma_c[\ell].status = \boldsymbol{C}$ then $v = \sigma_c[\ell].word$.
**(2)** $\forall \tau \in \{c, p\}.\forall i \in [k_\tau].\sigma'_\tau[i].status = \sigma_\tau[i].status \oplus ir_\tau[i]$.
**(3)** $\forall \tau \in \{c, p\}.\forall i \in [k_\tau]$.

$$\sigma'_\tau[i].word = \begin{cases} u :: v & (\text{if } \boldsymbol{O} \Rightarrow_{\tau,i} \boldsymbol{O}) \\ \sigma_\tau[i].word & (\text{if } _ \Rightarrow_{\tau,i} \boldsymbol{C}) \\ v & (\text{if } \boldsymbol{C} \Rightarrow_{\tau,i} \boldsymbol{O}) \end{cases}$$

where $\sigma_\tau[i].status \Rightarrow_{\tau,i} \sigma'_\tau[i].status$, $bts = \sigma_p[j].word$ where $j = argmax_{j \in J}|\sigma_p[j].word|$ if $J = \{j \in [k_p] \mid \boldsymbol{O} \Rightarrow_{p,j} \boldsymbol{C}\} \neq \emptyset$ and otherwise $bts = \epsilon$, and $u = \sigma_\tau[i].word/bts$ if $bts$ is a suffix of $\sigma_\tau[i].word$ and otherwise $u = \epsilon$.
**(4)** $w' = (bts :: w)\backslash v$.

We write $(q, w, \sigma_c, \sigma_p) \to^* (q', w', \sigma'_c, \sigma'_p)$ for $(q, w, \sigma_c, \sigma_p) \to \cdots \to (q', w', \sigma'_c, \sigma'_p)$. We give an intuitive reading of the definition. Roughly, $v$ is the string to be consumed by the step, and (1) says that if $\ell \in \Sigma_\epsilon$ then $\ell$ is consumed, and otherwise $\ell$ is a capturing-group index (i.e., $\ell \in [k_c]$) and the string stored at the corresponding memory, i.e., $\sigma_c[\ell].word$, is consumed provided that the memory is closed. (2) and (3) stipulate how the statuses and strings of the memories are updated, respectively. Importantly, (3) defines the *backtrack string bts* to be used for backtracking caused by a closure of a positive lookahead memory (if any happens in the step). Namely, *bts* is set to be the longest string stored in positive lookahead memories closed by the step ($bts = \epsilon$ if no such closures happen), and is used in (3) to reset the content of the memories that are open and remain so (i.e., those satisfying $\boldsymbol{O} \Rightarrow_{\tau,i} \boldsymbol{O}$) by right-division (cf. the definition of $u$). The string contents remain unchanged for memories that are or remain closed by the step (i.e., those satisfying $_ \Rightarrow_{\tau,i} \boldsymbol{C}$), and for the rest of the memories, the consumed string $v$ is appended to their strings. (4) defines $w'$, which is the input string in the post configuration (i.e., the string to be consumed in the continuation of the step), by prepending *bts* to the previous configuration's input string $w$ to account for any backtracking that happens in the step, and consuming $v$ by left-division. We remark that, for any configuration reachable from an initial configuration (see below), $\sigma_p[i].word$ is a prefix of $\sigma_p[j].word$ or vice versa for any $i, j \in [k_c]$, thus ensuring that *bts* is uniquely determined.

An *initial configuration* is $(q_0, w, \sigma_{c,0}, \sigma_{p,0})$, where $\sigma_{\tau,0}[i] = (\epsilon, \boldsymbol{C})$ for all $i \in [k_\tau]$ and $\tau \in \{c, p\}$. That is, every memory is initially closed and stores the empty string. A *run* of a PLMFA $M$ is a sequence $\pi$ such that $\pi[1]$ is an initial configuration and $\pi[i] \xrightarrow[M]{\ell} \pi[i+1]$ for all $1 \leq i < |\pi|$. A run $\pi$ is *accepting* if $\pi[|\pi|] = (q, \epsilon, _, _)$ for some $q \in F$. A string $w$ is *accepted* by $M$ if $M$ has an accepting run. The language of $M$, denoted by $L(M)$, is the set of strings accepted by $M$. That is, $L(M) = \{w \in \Sigma^* \mid (q_0, w, \sigma_{c,0}, \sigma_{p,0}) \to^* (q, \epsilon, \sigma_c, \sigma_p) \wedge q \in F\}$. We note that when $k_p = 0$, a PLMFA($k_c, k_p$) is a $k_c$-*memory automaton* (MFA($k_c$) or simply MFA if $k_c$ is irrelevant) introduced by Schmid [14].

▶ **Example 16.** As an example, consider a run of the PLMFA $M$ shown in Fig. 3, which is equivalent to the rewbl$_p$ `(?=(`$_1$`.*)`$_1$`z).\1` described in Example 4. In the figure, the (resp. double) circles represent (resp. accepting) states. The arrows represent transitions and the words on the labels represent labels on the transitions except for (?=, $(_1$, $)_1$, and ). The arrow with the word (?= (resp. )) represents the transition $\delta(q_1, \epsilon) \ni (q_2, \diamond, \mathtt{o})$ (resp. $\delta(q_5, \epsilon) \ni (q_6, \diamond, \mathtt{c})$). Additionally, the arrow with the word $(_1$ (resp. $)_1$) represents the transition $\delta(q_2, \epsilon) \ni (q_3, \mathtt{o}, \diamond)$ (resp. $\delta(q_3, \epsilon) \ni (q_4, \mathtt{c}, \diamond)$). The rewbl$_p$ expression contains just one backreference and positive lookahead. For simplicity, we abbreviate the lists of memories $[(x, \boldsymbol{s})]$ as $(x, \boldsymbol{s})$.

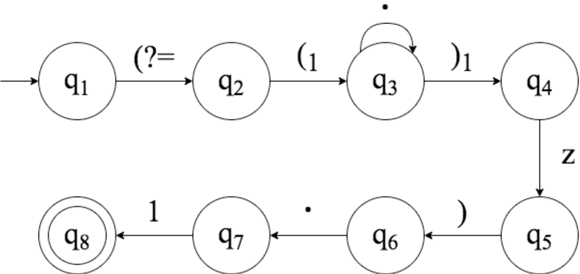



**Figure 3** A PLMFA equivalent to the rewbl$_p$ expression `(?=(`$_1$`·*)`$_1$`z)·\1`.

Given an input string $w$=zz, the run of $M$ is as follows: The run begins with the initial configuration $(q_0, \texttt{zz}, (\epsilon, \boldsymbol{C}), (\epsilon, \boldsymbol{C}))$. First, the initial configuration changes to $(q_3, \texttt{zz}, (\epsilon, \boldsymbol{O}), (\epsilon, \boldsymbol{O}))$ by applying the transitions $\delta(q_1, \epsilon) \ni (q_2, \diamond, \texttt{o})$ and $\delta(q_2, \epsilon) \ni (q_3, \texttt{o}, \diamond)$ in this order. The transition $\delta(q_1, \epsilon) \ni (q_2, \diamond, \texttt{o})$ opens the positive-lookahead memory and the configuration changes to $(q_2, \texttt{zz}, (\epsilon, \boldsymbol{C}), (\epsilon, \boldsymbol{O}))$. The transition $\delta(q_2, \epsilon) \ni (q_3, \texttt{o}, \diamond)$ opens the capturing-group memory and the configuration changes to $(q_3, \texttt{zz}, (\epsilon, \boldsymbol{O}), (\epsilon, \boldsymbol{O}))$. Next, the transitions $\delta(q_3, \cdot) \ni (q_3, \diamond, \diamond)$, $\delta(q_3, \epsilon) \ni (q_4, \texttt{c}, \diamond)$, and $\delta(q_4, \texttt{z}) \ni (q_5, \diamond, \diamond)$ are applied in this order. For the first transition, the configuration changes to the configuration $(q_3, \texttt{z}, (\texttt{z}, \boldsymbol{O}), (\texttt{z}, \boldsymbol{O}))$ by consuming a character z. For the second transition, the configuration changes to the configuration $(q_4, \texttt{z}, (\texttt{z}, \boldsymbol{C}), (\texttt{z}, \boldsymbol{O}))$ by closing the capturing-group memory. For the third transition, the configuration changes to the configuration $(q_5, \epsilon, (\texttt{z}, \boldsymbol{C}), (\texttt{zz}, \boldsymbol{O}))$ by consuming a character z.

Then, the configuration changes to $(q_6, \texttt{zz}, (\texttt{z}, \boldsymbol{C}), (\texttt{zz}, \boldsymbol{C}))$ by applying the transition $(q_5, \epsilon) \ni (q_6, \diamond, \texttt{c})$. The transition closes the positive-lookahead memory and therefore it simulates the backtracking behavior of the positive lookahead, i.e., it prepends the word of the positive-lookahead memory zz to the input string. Finally, the configuration changes to $(q_8, \epsilon, (\texttt{z}, \boldsymbol{C}), (\texttt{zz}, \boldsymbol{C}))$ by applying the transitions $\delta(q_6, \cdot) \ni (q_7, \diamond, \diamond)$ and $\delta(q_7, 1) \ni (q_8, \diamond, \diamond)$ in this order. The transition $\delta(q_6, \cdot) \ni (q_7, \diamond, \diamond)$ consumes a character z and the configuration changes to $(q_7, \texttt{z}, (\texttt{z}, \boldsymbol{C}), (\texttt{zz}, \boldsymbol{C}))$. Next, the current state $q_7$ has the transition of the backreference \1, i.e., $\delta(q_7, 1) \ni (q_8, \diamond, \diamond)$. The transition tries to match the captured string, i.e., z, with the current input string. Since the match succeeds, the configuration changes to $(q_8, \epsilon, (\texttt{z}, \boldsymbol{C}), (\texttt{zz}, \boldsymbol{C}))$. Now, the current state $q_8$ is an accepting state and the current input string is $\epsilon$, $w$ is accepted by $M$.

▶ Remark 17. As seen above, capturing-group memories and positive-lookahead memories exhibit an interesting *symmetry*: at their use, the content of a capturing-group (resp. positive-lookahead) memory is left-divided from (resp. prepended to) the input string, and appended to (resp. right-divided from) the strings stored in memories. The symmetry is imperfect because positive-lookahead memories do not have "triggers" corresponding to backreferences of capturing-group memories and a use of a positive-lookahead memory is always synchronous with its closure. A perfect symmetry can be obtained by extending PLMFA with a new kind of transitions that trigger positive-lookahead memory uses, disassociating them from closures. The extension certainly does not decrease the expressive power of PLMFA and we conjecture that it will strictly increase the expressive power.

We define conditions on PLMFA that correspond to RMRL and NUR of rewbl (cf. Sec. 2). Note that we do not define conditions on PLMFA that correspond to NSR of rewbl because PLMFAs already satisfy such a condition, i.e., PLMFAs do not allow to refer to the memory

whose status is open. For convenience, we simply call these conditions RMRL and NUR. Informally, a PLMFA satisfies RMRL if for all capturing-group memory (index) $i$ and a state $q$ from which the $i$th memory can be backreferenced, there exist a unique pair of transitions $a$ and $b$ that opened and closed the $i$th memory respectively so that the content of the $i$th memory when the computation reaches $q$ is what was recorded between $a$ and $b$. Intuitively, the pair of transitions correspond to the capturing group opening $(_i$ and closing $)_i$ of rewbl. Next, we formalize RMRL for PLMFA. For a configuration $\varpi$ and $i \in [k_c]$, let us write $\mathit{status}(\varpi, i)$ for the status of the $i$th capturing-group memory of $\varpi$, i.e., $\sigma_c[i].\mathit{status}$ where $\varpi = (_, _, \sigma_c, _)$.

▶ **Definition 18** (Opening and Closing Transitions Pair). For $q \in Q$ and $i \in [k_c]$, a pair of transitions $(\delta(p', \ell') \ni (q', ir'_c, ir'_p), \delta(p'', \ell'') \ni (q'', ir''_c, ir''_p)) = (a, b)$ is called an *opening-and-closing-transitions pair* of index $i$ at state $q$ if there exist a run $\pi$ and $1 \leq j_1 < j_2 < |\pi|$ such that (1) the step from $\pi[j_1]$ to $\pi[j_1 + 1]$ takes the transition $a$ and $\mathit{status}(\pi[j_1], i) = \boldsymbol{C}$, (2) the step from $\pi[j_2]$ to $\pi[j_2 + 1]$ takes the transition $b$ and $\mathit{status}(\pi[j_2 + 1], i) = \boldsymbol{C}$, (3) for all $j_1 < l \leq j_2$, $\mathit{status}(\pi[l], i) = \boldsymbol{O}$, and for all $j_2 < l \leq |\pi|$, $\mathit{status}(\pi[l], i) = \boldsymbol{C}$, and (4) the state of $\pi[|\pi|]$ is $q$. We define $\mathit{RefSet}_{M,i}(q)$ (or $\mathit{RefSet}_i(q)$ if there is no danger of ambiguity) as the set of opening-and-closing-transitions pairs of $i$ at $q$ on $M$.

▶ **Definition 19** (RMRL-PLMFA). A PLMFA$(k_c, k_p)$ $M = (Q, \delta, q_0, F)$ is called *restricted may repeat labels* (RMRL) if for all $(q, i) \in Q \times [k_c]$ such that $\delta(q, i) \neq \emptyset$, $|\mathit{RefSet}_i(q)| \leq 1$.

Next, we define NUR for PLMFA. Informally, a PLMFA satisfies NUR if no capturing-group memory can be backreferenced without capturing a word. Formally, for $(q, i) \in Q \times [k_c]$, we say that $q$ is *assigned* with respect to index $i$ on $M$, written $\mathit{Assigned}_M(q, i)$ (or $\mathit{Assigned}(q, i)$ if there is no danger of ambiguity), if for all runs $\pi$ such that the state of $\pi[|\pi|]$ is $q$, there exists $1 \leq j < |\pi|$ such that $\mathit{status}(\pi[j], i) = \boldsymbol{O}$ and $\mathit{status}(\pi[j + 1], i) = \boldsymbol{C}$.

▶ **Definition 20** (NUR-PLMFA). A PLMFA$(k_c, k_p)$ $M = (Q, \delta, q_0, F)$ is *no unassigned reference (NUR)* if for all $(q, i) \in Q \times [k_c]$ such that $\delta(q, i) \neq \emptyset$, $\mathit{Assigned}(q, i) = \mathit{true}$.

In what follows, we assume that PLMFAs satisfy RMRL and NUR.

## 4.2 Normal Forms and Nested Forms

We show that a PLMFA can be converted into certain forms. The *normal form* enforces two restrictions: (1) only $\epsilon$ transitions can change memory statuses and at most one status of the memory at a time, and (2) no transitions open (resp. close) a memory that is already opened (resp. closed).

▶ **Definition 21** (Normal Form). A PLMFA is in *normal form* if the following properties are satisfied. For every transition $\delta(q, \ell) \ni (q', ir_c, ir_p)$, $\tau \in \{c, p\}$, and $j \in [k_\tau]$, (1) if $ir_\tau[j] \neq \diamond$, then $\ell = \epsilon$ and $ir_{\tau'}[l].\mathit{status} = \diamond$ for all $(\tau', l) \in \{c, p\} \times [k_{\tau'}]$ such that $(\tau', l) \neq (\tau, j)$, and (2) there is no run $\pi$ such that $\pi[|\pi|] = (_, _, \sigma_c, \sigma_p)$ where $ir_\tau[j] = \mathsf{o}$ and $\sigma_\tau[j].\mathit{status} = \boldsymbol{O}$ or $ir_\tau[j] = \mathsf{c}$ and $\sigma_\tau[j].\mathit{status} = \boldsymbol{C}$.

▶ **Lemma 22.** *Any PLMFA $M$ can be converted to a normal form PLMFA $M'$ such that $L(M) = L(M')$.*

The proof is by adopting an analogous conversion of [14] and works by extending the states of $M$ to record memory statuses and splitting simultaneous memory updates to multiple transitions. We remark that the conversion preserves RMRL and NUR.

Next, we define *nested form* which enforces that there are no *overlaps* in any runs. A run $\pi$ is said to have an overlap if there exist $1 \leq j_1 < j_2 < j_3 < j_4 < |\pi|$ such that some memory is opened and closed respectively at the $j_1$th and the $j_3$th steps and some memory (possibly the same) is opened and closed respectively at the $j_2$th step and the $j_4$th step. There are four types of overlaps, *cc*, *cp*, *pc*, and *pp*, depending on the types of the first and the second memories (e.g., *cp*-overlap is when the first memory is capturing-group and the second is positive-lookahead). Intuitively, an overlap corresponds to an invalid expression that has an overlap of capturing groups or positive lookaheads. For example, *cc*-overlap corresponds to invalid expressions $({}_i r_1 ({}_j r_2)_i r_3)_j$ and *pc*-overlap corresponds to invalid expressions $(?{=}r_1({}_i r_2)r_3)_i$.

▶ **Definition 23** (Nested Form)**.** A PLMFA $M$ is in *nested form* if there are no overlaps in any runs on $M$.

We show that we can transform a PLMFA to the nested form.

▶ **Lemma 24.** *Any PLMFA $M$ can be converted to a normal and nested form PLMFA $M'$ such that $L(M) = L(M')$.*

The proof is by adding new transitions that close the fragments of the memories which are open before opening the transitions that cause overlaps and open the next fragments of the memories after that. For cc-overlaps, the conversion coincides with the analogous one for MFAs [14]. In what follows, without loss of generality, we assume that PLMFAs are in normal and nested form.

## 4.3 From Rewbl$_p$ to PLMFA

We show that given a rewbl$_p$ expression $r$, we can construct a PLMFA $M$ such that $L(r) = L(M)$. The construction of the PLMFA extends the standard Thompson construction from a pure regular expression to an NFA [15] with backreferences and lookaheads in a mostly straightforward manner. For space, the construction and the proof of correctness is omitted.

▶ **Theorem 25.** *Let a rewbl$_p$ $r$ include $k_c$ capturing groups and $k_p$ positive lookaheads. Then, there exists a PLMFA($k_c, k_p$) $M$ such that $L(r) = L(M)$.*

## 4.4 From PLMFA to Rewbl$_p$

Now, we show that given a PLMFA $M$, we can construct a rewbl$_p$ expression $r$ such that $L(M) = L(r)$. We first give the conversion and then show the correctness. For space, we only show the correctness of the language equivalence and omit the fact that the conditions NUR, NSR, and RMRL are satisfied by the resulting rewbl$_p$. However, they can be proved similarly to the language equivalence.

The conversion, referred to as *PtoR*, is inspired by that of MFAs to rewbs [14]. The idea of the conversion is to use the nested relation of PLMFAs. Since PLMFAs are nested form, the transitions of the opening-and-closing-transitions pairs have a nested structure, i.e., they form a directed acyclic graph (DAG). Therefore, we can iteratively convert (sub)automaton corresponding to the part of the given PLMFA delimited by each such pairs to the rewbl$_p$ expression in a topological order starting from the pairs that nest nothing. Each step of the conversion makes an *extended PLMFA* (ePLMFA) whose labels are rewbl$_p$ expressions. That is, labels $\ell$ on transitions of PLMFAs are treated as rewbl$_p$ expressions $\ell$ of ePLMFAs if

$\ell \in \Sigma_\epsilon$. Additionally, if $\ell = i \in [k_c]$ on the transitions of PLMFAs, $\ell = \backslash i$ on the transitions of ePLMFAs. For an ePLMFA $M$ and a rewbl$_p$ expression $\ell$, a step of computation of $M$ reading $\ell$ is defined as follows: $(q, w, \sigma_c, \sigma_p) \xrightarrow{\ell} (q', w', \sigma'_c, \sigma'_p)$ iff either $\ell = \epsilon$ and $(q, w, \sigma_c, \sigma_p) \xrightarrow{\epsilon} (q', w', \sigma'_c, \sigma'_p)$ according to Def. 15, or $\ell \neq \epsilon$ and there exist a transition $\delta(q, \ell) \ni (q', \diamond^{k_c+k_p})$ and steps of computations from $(q'_0, w, \sigma_c, \sigma_p)$ to $(q'_f, w', \sigma'_c, \sigma'_p)$ of $M' = (_, _, q'_0, \{q'_f\})$ obtained from $\ell$ by the construction from a rewbl$_p$ expression to a PLMFA mentioned in Sec. 4.3.

We also extend the NUR and RMRL conditions for ePLMFAs as follows. For all transition, we reconstruct PLMFA from the rewbl$_p$ label by applying the construction mentioned in Sec. 4.3 and replace the transition with the PLMFA by replacing it with $\epsilon$-labeled $\diamond$-only transitions to and from the initial and the final state of the PLMFA. We say that an ePLMFA satisfies NUR and RMRL if the reconstructed PLMFA satisfies NUR and RMRL, respectively.

We define the nesting relation. Firstly, for an ePLMFA and $\texttt{inst} \in \{\texttt{o}, \texttt{c}\}$, we define $Q_{\tau,i,\texttt{inst}}$ as the set of states $q$ such that $\delta(q, _) \ni (_, ir_c, ir_p)$ where $ir_\tau[i] = \texttt{inst}$. Let $\Phi = \{(\tau, i, q, q') \mid \tau \in \{c, p\} \wedge i \in [k_\tau] \wedge q \in Q_{\tau,i,\texttt{o}} \wedge q' \in Q_{\tau,i,\texttt{c}}\}$. The *nesting relation* $\prec \subseteq \Phi \times \Phi$ is defined as follows: $(\tau_1, i_1, q_1, q'_1) \prec (\tau_2, i_2, q_2, q'_2)$ iff there exist a run $\pi$ and $s < t < u < v < \pi[|\pi|]$ such that the step from $\pi[s]$ to $\pi[s+1]$ takes a transition $\delta(q_2, _) \ni (_, ir_c, ir_p)$ where $ir_{\tau_2}[i_2] = \texttt{o}$, the step from $\pi[t]$ to $\pi[t+1]$ takes a transition $\delta(q_1, _) \ni (_, ir_c, ir_p)$ where $ir_{\tau_1}[i_1] = \texttt{o}$, the step from $\pi[u]$ to $\pi[u+1]$ takes a transition $\delta(q'_1, _) \ni (_, ir_c, ir_p)$ where $ir_{\tau_1}[i_1] = \texttt{c}$, and the step from $\pi[v]$ to $\pi[v+1]$ takes a transition $\delta(q'_2, _) \ni (_, ir_c, ir_p)$ where $ir_{\tau_2}[i_2] = \texttt{c}$.

For an ePLMFA $M$, the procedure of *PtoR*$(M)$ is defined as follows. Let us initialize $\Delta : \Phi \to \mathcal{P}(\Phi)$ as follows: $\Delta(\tau_1, i_1, q_1, q'_1) = \{(\tau_2, i_2, q_2, q'_2) \mid (\tau_2, i_2, q_2, q'_2) \prec (\tau_1, i_1, q_1, q'_1)\}$. We iteratively update $M$, $\Phi$, and $\Delta$ by the following steps.

1. Find $(\tau, i, q, q') \in \Phi$ such that $\Delta(\tau, i, q, q') = \emptyset$. Let $\delta(q, _) \ni (q_s, _)$ (resp. $\delta(q', _) \ni (q_e, _)$) be the opening (resp. closing) transition of $(\tau, i, q, q')$. Then, construct an ePLMFA $M'$ from $M$ by replacing the initial state and set of accepting states with $q_s$ and $\{q'\}$, respectively, and deleting all transitions that open or close memories.
2. Convert $M'$ to a rewbl$_p$ expression $r$ using the standard state elimination method.
3. Delete $(\tau, i, q, q')$ from $\Phi$ and (the domain and the range of) $\Delta$, and add the transition $\delta(q, ({}_i r)_i) \ni (q_e, \diamond^{k_c+k_p})$ if $\tau = c$ and otherwise $\delta(q, (?{=}r)) \ni (q_e, \diamond^{k_c+k_p})$ to $M$.
4. Repeat steps 1 to 3 until convergence.
5. Delete all transitions that open or close memories from $M$.
6. Convert $M$ to a rewbl$_p$ expression $r$ by the state elimination method and return $r$.

At step 1, we can find such a tuple $(\tau, i, q, q')$ since it is in nested form. The state elimination method used in steps 2 and 6 is a straightforward adoption of the standard state elimination method (see, e.g., [13]) that interprets the rewbl$_p$ expression that appear as labels as ordinary regular expressions.

We proceed to the proof of correctness.

▶ **Definition 26.** The *language* of an ePLMFA $M = (Q, \delta, q_0, F)$ parameterized by a string $w \in \Sigma^*$, a capturing-group memory $\sigma_c$, and a positive-lookahed memory $\sigma_p$, denoted by $L(M, w, \sigma_c, \sigma_p)$, is defined as

$$L(M, w, \sigma_c, \sigma_p) \quad = \quad \{(w', \sigma'_c, \sigma'_p) \mid \exists q' \in F.\ (q_0, w, \sigma_c, \sigma_p) \to^* (q', w', \sigma'_c, \sigma'_p)\}.$$

For an ePLMFA $M = (Q, \delta, q_0, F)$, we write $M(q', F')$ for the ePLMFA $M' = (Q, \delta, q', F')$ where $q' \in Q$ and $F' \subseteq Q$. We show that eliminating a state from an ePLMFA by applying one step of the state elimination method does not change the parameterized language of the ePLMFA. The state elimination method eliminates a state by deleting and adding transitions as shown in Fig. 4.

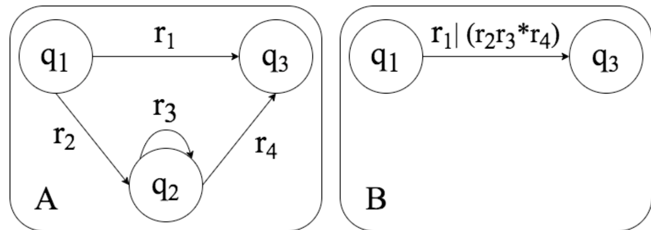



**Figure 4** (A) Before eliminating the state $q_2$. (B) After eliminating the state $q_2$.

▶ **Lemma 27.** *Let $M = (Q, \delta, q_0, F)$ be an ePLMFA$(k_c, k_p)$ such that for all transitions $\delta(q, \ell) \ni (q', ir_c, ir_p)$, $i \in [k_c]$, and $j \in [k_p]$, $ir_c[i] = \diamond$ and $ir_p[j] = \diamond$. Additionally, let $M' = (Q', \delta', q'_0, F')$ be the ePLMFA obtained by eliminating a state in $Q$ from $M$ by the state elimination method. Then, for all $w \in \Sigma^*$, $\sigma_c$, and $\sigma_p$, $L(M, w, \sigma_c, \sigma_p) = L(M', w, \sigma_c, \sigma_p)$.*

**Proof.** Let us assume that $q_1$ and $q_3$ are in $Q$ and $Q'$, $q_2 \in Q$ is the state eliminated by the state elimination method as shown in Fig. 4. Since the only difference between $M$ and $M'$ comes from the state $q_2$, it suffices to show that $L(M(q_1, \{q_3\}), w, \sigma_c, \sigma_p) = L(M'(q_1, \{q_3\}), w, \sigma_c, \sigma_p)$. It is immediate from the construction. ◀

▶ **Theorem 28.** *For an ePLMFA $M$, let $M_j$ be the ePLMFA obtained after the $j$th iteration of steps 1 to 3 of PtoR$(M)$. We assume $M_j = M$ if $j = 0$. Then, for all $w \in \Sigma^*$, $\sigma_c$, and $\sigma_p$, $L(M, w, \sigma_c, \sigma_p) = L(M_j, w, \sigma_c, \sigma_p)$.*

**Proof.** The proof is by induction on the number of the iteration $j$ of step 1 to 3. The base step $j = 0$ is immediate since $M_j = M$. For the induction step, let $j > 0$. The inductive hypothesis is that $L(M, w, \sigma_c, \sigma_p) = L(M_j, w, \sigma_c, \sigma_p)$ holds for all $j$, $w$, $\sigma_c$, and $\sigma_p$. We then consider $M_{j+1}$. We show $L(M_j, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$. As described in the step 3, for the ePLMFA $M_j = (Q, \delta, q_0, F)$, $M_{j+1} = (Q, \delta \cup \{t\}, q_0, F)$ where $t = \delta(q, r') \ni (q_e, \diamond^{k_c+k_p})$. Recall that $r' = ({}_i r)_i$ or $(?{=}r)$ and $r$ is the rewbl$_p$ expression obtained at step 2. For this, $L(M_j, w, \sigma_c, \sigma_p) \subseteq L(M_{j+1}, w, \sigma_c, \sigma_p)$ is immediate. For $L(M_{j+1}, w, \sigma_c, \sigma_p) \subseteq L(M_j, w, \sigma_c, \sigma_p)$, it suffices to show that for all $w$, $\sigma_c$, and $\sigma_p$, if there exists $(q, w, \sigma_c, \sigma_p) \xrightarrow{r'} (q_e, w', \sigma'_c, \sigma'_p)$ on $M_{j+1}$, then there exists $(q, w, \sigma_c, \sigma_p) \to^* (q_e, w', \sigma'_c, \sigma'_p)$ on $M_j$. We assume that there exists $\pi = (q, w, \sigma_c, \sigma_p) \xrightarrow{r'} (q_e, w', \sigma'_c, \sigma'_p)$ on $M_{j+1}$. The computation of the transition whose label is $r'$ is defined by that of the ePLMFA $M_{r'}$ obtained from $r'$ by the construction mentioned in Sec. 4.3. If $r' = ({}_i r)_i$, $M_{r'} = (Q_{r'}, \delta_{r'}, q, \{q_e\})$ where

- $Q_{r'} = \{q, q_e\} \cup Q_r$; and
- $\delta_{r'} = \delta_r \cup \{((q, \epsilon), \{(q_{0,r}, ir_c, \diamond^{k_p})\})\} \cup \{((q_{F,r}, \epsilon), \{(q_e, ir'_c, \diamond^{k_p})\})\}$ where $ir_c[k] = \mathsf{o}$ and $ir'_c[k] = \mathsf{c}$ if $k = i$ and otherwise $ir_c[k] = \diamond$ and $ir'_c[k] = \diamond$ for $k \in [k_c]$

with the ePLMFA for $r$ $M_r = (Q_r, \delta_r, q_{0,r}, \{q_{F,r}\})$. By the transitions in $\delta_{r'}$, $\pi = (q, w, \sigma_c, \sigma_p) \to (q_{0,r}, w, \sigma''_c, \sigma_p) \xrightarrow{r} (q_{F,r}, w', \sigma'''_c, \sigma'_p) \to (q_e, w', \sigma'_c, \sigma'_p)$ where $\sigma''_c[k] = (\epsilon, \boldsymbol{O})$ and $\sigma'''_c[k] = (\sigma'_c[k].word, \boldsymbol{O})$ if $k = i$ and otherwise $\sigma''_c[k] = \sigma_c[k]$ and $\sigma'''_c[k] = \sigma'_c[k]$. We focus on the computation $(q_{0,r}, w, \sigma''_c, \sigma_p) \xrightarrow{r} (q_{F,r}, w', \sigma'''_c, \sigma'_p)$. The label $r$ is constructed from $M'$ by the state elimination method at step 2. Additionally, by Lemma 27, the state elimination method preserves the parameterized language equivalence. For this, there exists $(q_s, w, \sigma''_c, \sigma_p) \to^* (q', w', \sigma'''_c, \sigma'_p)$ on $M'$. Since the transition function of $M_j$ includes all transitions of $M'$, there also exists $(q_s, w, \sigma''_c, \sigma_p) \to^* (q', w', \sigma'''_c, \sigma'_p)$ on $M_j$. As described at step 1, $M_j$ has a transition from $q$ to $q_s$ that opens the $i$th capturing-group memory

and a transition from $q'$ to $q_e$ that closes the $i$th capturing-group memory. Therefore, there exists $(q, w, \sigma_c, \sigma_p) \to (q_s, w, \sigma_c'', \sigma_p) \to^* (q', w', \sigma_c''', \sigma_p') \to (q_e, w', \sigma_c', \sigma_p')$. For this, $L(M_j, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$. By the inductive hypothesis, $L(M_j, w, \sigma_c, \sigma_p) = L(M, w, \sigma_c, \sigma_p)$. Thus, $L(M, w, \sigma_c, \sigma_p) = L(M_{j+1}, w, \sigma_c, \sigma_p)$ for the case $r' = (_i r)_i$. The case of $r' = (?{=}r)$ is analogous. ◀

Now, we obtain our main result.

▶ **Theorem 29.** *Let $M$ be an ePLMFA. Then, $L(M) = L(r)$ where $r = PtoR(M)$.*

**Proof.** Let $M'$ be the ePLMFA at the last step of the state elimination method at step 6 of *PtoR*. Then, $M' = (\{q, q'\}, \{((q, r), \{(q', \diamond^{k_c + k_p})\})\}, q, \{q'\})$. By Theorem 25, $L(M') = L(r)$. By Theorem 28, for all $w \in \Sigma^*$, $L(M', w, \sigma_{c,0}, \sigma_{p,0}) = L(M, w, \sigma_{c,0}, \sigma_{p,0})$, i.e., $L(M') = L(M)$. Thus, $L(M) = L(M') = L(r)$. ◀

As a corollary of Theorem 25 and Theorem 29, we obtain the following result.

▶ **Corollary 30.** *The expressive power of PLMFA is equivalent to that of $rewbl_p$.*

## 5 Related Work

Among the major extensions employed by modern real-world regular expression engines are backreferences and lookaheads. However, the previous works on formal language theory have studied the two features mostly in isolation, and to the best of our knowledge, our work is the first formal study of regular expressions extended with both features. Next, we discuss previous works that studied the features in isolation.

Prior works by Morihata and Berglund et al. [12, 3] showed that extending regular expressions by lookaheads does not enhance their expressive power. Their proofs are by a translation to boolean finite automata [4] whose expressive power is regular. However, adopting such an approach to defining an equivalent automata is difficult in the presence of backreferences because boolean automata express lookaheads by running several states simultaneously without backtracking, while the combination of lookaheads and backreferences intrinsically requires backtracking. For example, to match against `(?=·(·)`$_1$`)\1·`, a boolean approach would run `(?=·(·)`$_1$`)` and `\1·` simultaneously, but then the automaton would get stuck while trying to process the backreference `\1` as it is unassigned at that point. By constrast, our PLMFA uses the novel positive-lookahead memories to store enough information to simulate the backtracking behavior of positive lookaheads.

A formal study of regular expressions with backreferences (rewb) dates back to the seminal work by Aho [1]. More recently, a formal semantics and a pumping lemma were given by Câmpeanu et al. [5]. Berglund and van der Merwe [2] showed that different variants of backreference semantics give rise to differences in expressive powers. Our works adopts and formalizes the no-label-repetitions (NLR) and no-unassigned-reference (NUR) semantics which is also used in [5, 6]. Schmid [14] proposed MFA, and showed that the expressive power of the automata is equivalent to that of rewb. Our PLMFA builds on MFA and extends it with positive-lookahead memories to handle positive lookaheads. As remarked before (cf. Remark 17), our positive-lookahead memory exhibits an interesting symmetry to the capturing-group memory of MFA.

## 6 Conclusion

We have studied the expressive powers of regular expressions with the popular backreferences and lookaheads extensions. We have shown that extending rewb by positive or negative lookaheads enhance their expressive power. Additionally, we have presented language-theoretic properties of rewb extended by the two forms of lookaheads, and have presented a new class of automata called PLMFA that is equivalent in expressive power to rewbl$_p$. We have introduced a new kind of memories called a positive-lookahead memory, which is almost perfectly symmetric to capturing-group memory of MFA, as a key component of PLMFA.

Despite the popularity of the backreference and lookaheads extensions in practice, to our knowledge, our work is the first formal study on regular expressions with both extensions. We hope that our results pave the way for more work on the topic.

## References


1. Alfred V. Aho. Chapter 5 - algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 255–300. Elsevier, Amsterdam, 1990. `doi:10.1016/B978-0-444-88071-0.50010-2`.
2. Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*, pages 30–41. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017. URL: `http://www.stringology.org/event/2017/p04.html`.
3. Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. Regular expressions with lookahead. *J. Univers. Comput. Sci.*, 27(1):324–340, 2021. `doi:10.3897/jucs.66330`.
4. Janusz A. Brzozowski and Ernst L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980. URL: `http://dblp.uni-trier.de/db/journals/tcs/tcs10.html#BrzozowskiL80`.
5. Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. `doi:10.1142/S012905410300214X`.
6. Benjamin Carle and Paliath Narendran. On extended regular expressions. In Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2009. `doi:10.1007/978-3-642-00982-2_24`.
7. Ashok K. Chandra and Larry J. Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 98–108, 1976. `doi:10.1109/SFCS.1976.4`.
8. N. Chida and T. Terauchi. Repairing dos vulnerability of real-world regexes. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy, SP 2022*, pages 1049–1066, Los Alamitos, CA, USA, May 2022. IEEE Computer Society. `doi:10.1109/SP46214.2022.00061`.
9. S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*. 1956.
10. Vladimir Komendantsky. Matching problem for regular expressions with variables. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2012. `doi:10.1007/978-3-642-40447-4_10`.
11. Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of javascript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 425–438, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3314221.3314645`.

12 Akimasa Morihata. Translation of regular expression with lookahead into finite state automaton. *Computer Software*, 29(1):1_147–1_158, 2012. doi:10.11309/jssst.29.1_147.

13 Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, USA, 2009.

14 Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Inf. Comput.*, 249:1–17, 2016. doi:10.1016/j.ic.2016.02.003.

15 Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. doi:10.1145/363347.363387.

# Certified Decision Procedures for Two-Counter Machines

**Andrej Dudenhefner**
TU Dortmund, Germany

**Abstract**

Two-counter machines, pioneered by Minsky in the 1960s, constitute a particularly simple, universal model of computation. Universality of *reversible* two-counter machines (having a right-unique step relation) has been shown by Morita in the 1990s. Therefore, the halting problem for reversible two-counter machines is undecidable. Surprisingly, this statement is specific to certain instruction sets of the underlying machine model.

In the present work we consider two-counter machines (CM2) with instructions $\texttt{inc}_c$ (increment counter $c$, go to next instruction), $\texttt{dec}_c\, q$ (if counter $c$ is zero, then go to next instruction, otherwise decrement counter $c$ and go to instruction $q$). While the halting problem for CM2 is undecidable, we give a decision procedure for the halting problem for reversible CM2, contrasting Morita's result.

We supplement our result with decision procedures for uniform boundedness (is there a uniform bound on the number of reachable configurations?) and uniform mortality (is there a uniform bound on the number of steps in any run?) for CM2.

Termination and correctness of each presented decision procedure is certified using the Coq proof assistant. In fact, both the implementation and certification is carried out simultaneously using the tactic language of the Coq proof assistant. Building upon existing infrastructure, the mechanized decision procedures are contributed to the Coq library of undecidability proofs.

**2012 ACM Subject Classification** Theory of computation → Computability

**Keywords and phrases** constructive mathematics, computability theory, decidability, counter automata, mechanization, Coq

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.16

**Supplementary Material** *Software (Source Code)*: `https://github.com/uds-psl/coq-library-undecidability/tree/coq-8.15/theories/CounterMachines`

## 1 Introduction

In the early 1960s, Minsky has shown the universality of two-tape, read-only Turing machines [18]. As a result, two-counter machines (originally called "program-machines" [19, Table 11.1-1]) emerged as a particularly simple, universal model of computation. A two-counter machine stores data in two registers, each containing a natural number. While the particular instruction sets may vary (for an overview see [13, Section 2]), common machine instructions are: counter increment, counter decrement (possibly including a conditional jump), and counter zero test (including a conditional jump). Due to the arithmetically simple nature of machine instructions, two-counter machines are easily simulated by other machine models. This often leads to small, universal constructions [11]. Another prominent example, which relies on two-counter machines, is the nested simulation technique, invented by Hooper for Turing machine immortality [10, 12]. Besides the halting problem for two-counter machines, mortality and boundedness problems [14] constitute useful tools in the area of model checking.

In the research field of *reversible computing*, which considers "backward deterministic" computation, universality of reversible two-counter machines was shown by Morita [20]. By reversible simulation, this milestone result has immediate implications for other models of computation (for an overview see [21, 22]), and group theory [24].

© Andrej Dudenhefner;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 16; pp. 16:1–16:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Commonly, the instruction set of the underlying two-counter machine model is chosen suitably for the individual results. Key results such as (reversible) universality are transferred tacitly between instruction sets. Surprisingly, this is not always possible. The most prominent example is the decidability of the halting problem for “program-machines” with two registers (Remark 1), as originally given by Minsky.

▶ **Remark 1.** Consider the exact definition of “program-machines” given by Minsky [19, Table 11.1-1]. That is, lists of instructions from the following instruction set:

- set counter $c$ to zero, go to next instruction
- increment counter $c$, go to next instruction
- if counter $c$ is positive, then decrement counter $c$ and go to next instruction, otherwise go to instruction $n$
- halt

The halting problem for the above machine model with two counters is decidable[1]. Conditional control flow in the above machine model is based on failing counter decrement instructions. That is, if the value of one of the counters is zero. For example, given a program of length $n$, if both counters are larger than $n$, then (unable to go to any previously visited instruction) the program halts after at most $n$ steps. Therefore, for any run, values of at least one of the counters are drawn from a finite set. Overall, the above machine model with two counters is not universal.

In his argument, Minsky necessarily includes an additional, unconditional jump instruction [19, Chapter 14] in order to have a universal machine model with two counters.

In the present work we consider two-counter machines (CM2) as list of instructions from the instruction set: $\texttt{inc}_c$ (increment counter $c$, go to next instruction), $\texttt{dec}_c\, q$ (if counter $c$ is zero, then go to next instruction, otherwise decrement counter $c$ and go to instruction $q$). CM2, relying on its arguably minimal instruction set, plays a key role in mechanized undecidability results [8] (such as Hilbert’s tenth problem [15] and semi-unification [2]). The key difference to Minsky’s “program-machines” is that the conditional jump is on successful (instead of failed) counter decrement. In contrast to Remark 1, this instruction set suffices for an undecidable halting problem (Theorem 6). However, this instruction set does *not* suffice for an undecidable reversible halting problem (Theorem 21), which is our main result. Intuitively, conditional control flow for the above instruction set is too restricted in the reversible setting, and does not allow for nested loops. As a consequence, control flow for reversible CM2 can be modeled by a finite state automaton, resulting in a decision procedure for termination.

In addition, we consider boundedness and mortality problems for CM2. First, we contrast undecidability of total boundedness [14] (is for any configuration the number of reachable configurations finite?) for CM2 with a decision procedure for uniform boundedness (is there a uniform bound on the number of reachable configurations?). Second, we contrast undecidability of total mortality [10] for CM2 (does every run eventually halt?) with a decision procedure for uniform mortality (is there a uniform bound on the number of steps in any run?). While decidability of uniform mortality is known [12, Theorem 2], the more complex decidability of uniform boundedness is hitherto only hinted at [1, Remark 28]. Additionally, the decision algorithms provided in the present work use explicit upper bounds, and are well-suited for complexity-theoretic analysis.

---

[1] The certified decision procedure is mechanized as the computable Boolean function `decide : Mpm2 * Config -> bool` in `theories/MinskyMachines/MPM2_HALT_dec.v` in the Coq library of undecidability proofs [8].

Formal specification, termination certification, and verification of correctness of the presented decision procedures is carried out using the Coq proof assistant. Following the compelling argument by Forster [3], the Coq proof assistant is excellently positioned to argue about computability-theoretic properties of decision problems. For a predicate $P$ over a domain $X$, a *decision procedure* is a Boolean function $f : X \to \mathbb{B}$ such that for all $x \in X$ we have $(f(x) = \texttt{true}) \Leftrightarrow P(x)$. Any axiom-free implementation of a decision procedure $f$ in Coq entails a termination argument for $f$ on any input. As added benefit, the correctness proof of $f$ can be given and verified mechanically in Coq as part of the definition of $f$. Effective implementations of the individual decision procedures can be obtained using the `Extraction` framework [17]. Overall, the approach taken in the present work is well positioned in the intersection of computability theory and constructive mathematics.

The growing Coq library of undecidability proofs [8] already contains a plethora of negative and positive[2] computability results. Since CM2 is a prominent decision problem in the library, we build upon the existing infrastructure to contribute the decision procedures in the present work to the library.

**Organization.** The remainder of the present work is organized as follows.

**Section 2:** Preliminary definitions and properties of two-counter machines (CM2).
**Section 3:** Decision procedure for the halting problem for reversible CM2 (Theorem 21).
**Section 4:** Decision procedure for the uniform boundedness problem for CM2 (Theorem 40).
**Section 5:** Decision procedure for the uniform mortality problem for CM2 (Theorem 49).
**Section 6:** Remarks on the mechanization in the Coq proof assistant of the above decision procedures, and the contribution to the Coq library of undecidability proofs.
**Section 7:** Concluding remarks.

## 2 Two-Counter Machine Preliminaries

In this section we recollect the definition of two-counter machines (CM2, Definition 2) and the undecidability of the corresponding halting problem (Theorem 6). The main benefit of CM2 is its small, universal instruction set (cf. [13, Section 2]). As a result, CM2 allows for compact proofs, and plays a key role in mechanized undecidability results [8].

▶ **Definition 2** (Two-Counter Machine (CM2))**.** A *two-counter machine* $\mathcal{M}$ is a list of *instructions* of shape either $\texttt{inc}_0$, $\texttt{inc}_1$, $\texttt{dec}_0\, q$, or $\texttt{dec}_1\, q$, where $q \in \mathbb{N}$ is a *program index*.

A *configuration* of $\mathcal{M}$ is of shape $(p, (a, b))$, where $p \in \mathbb{N}$ is the current *program index* and $a, b \in \mathbb{N}$ are the current *counter values*.

The *step relation* of $\mathcal{M}$ on configurations, written $(\longrightarrow_{\mathcal{M}})$, is given by

- if $\texttt{inc}_0$ is the $p$-th instruction of $\mathcal{M}$, then $(p, (a, b)) \longrightarrow_{\mathcal{M}} (p + 1, (a + 1, b))$
- if $\texttt{inc}_1$ is the $p$-th instruction of $\mathcal{M}$, then $(p, (a, b)) \longrightarrow_{\mathcal{M}} (p + 1, (a, b + 1))$
- if $\texttt{dec}_0\, q$ is the $p$-th instruction of $\mathcal{M}$, then $(p, (0, b)) \longrightarrow_{\mathcal{M}} (p + 1, (0, b))$ and $(p, (a + 1, b)) \longrightarrow_{\mathcal{M}} (q, (a, b))$
- if $\texttt{dec}_1\, q$ is the $p$-th instruction of $\mathcal{M}$, then $(p, (a, 0)) \longrightarrow_{\mathcal{M}} (p + 1, (a, 0))$ and $(p, (a, b + 1)) \longrightarrow_{\mathcal{M}} (q, (a, b))$
- otherwise, we say that $(p, (a, b))$ *halts*

[2] For example, Spies and Forster [26] contrast undecidability of higher-order unification with a decision procedure for first-order unification in Coq.

The *reachability relation* of $\mathcal{M}$ on configurations, written $(\longrightarrow_{\mathcal{M}}^{*})$, is the reflexive, transitive closure of $(\longrightarrow_{\mathcal{M}})$. The transitive closure of $(\longrightarrow_{\mathcal{M}})$ is denoted by $(\longrightarrow_{\mathcal{M}}^{+})$.

A configuration $x$ is *terminating* in $\mathcal{M}$, if we have $x \longrightarrow_{\mathcal{M}}^{*} y$ for some halting configuration $y$.

The length of $\mathcal{M}$ is denoted by $|\mathcal{M}|$.

Comparing the above Definition 2 to Minsky's original definition [19, Table 11.1-1], the main difference is that the conditional jump is performed on a successful (instead of failed) counter decrement. In the setting with only two counters this difference is crucial for universality (Remark 1). Compared to Morita's definition [20, Definition 2.1], the above Definition 2 does not have separate (un)conditional jump instructions. As is shown in Section 3, in the reversible setting this difference allows for a decision procedure (Theorem 21) for the corresponding halting problem.

▶ **Example 3.** Consider $\mathcal{M} = [\mathtt{dec}_0\, 4, \mathtt{inc}_0, \mathtt{dec}_0\, 0]$. The configuration $(0,(a,b))$ for $a, b \in \mathbb{N}$ is terminating in $\mathcal{M}$ iff $a > 0$, as is shown below:

$$(0,(0,b)) \xrightarrow{\mathtt{dec}_0 4}_{\mathcal{M}} (1,(0,b)) \xrightarrow{\mathtt{inc}_0}_{\mathcal{M}} (2,(1,b)) \xrightarrow{\mathtt{dec}_0 0}_{\mathcal{M}} (0,(0,b)) \xrightarrow{\mathtt{dec}_0 4}_{\mathcal{M}} \cdots$$

$$(0,(a+1,b)) \xrightarrow{\mathtt{dec}_0 4}_{\mathcal{M}} (4,(a,b)) \text{ halts}$$

By definition, the step relation $(\longrightarrow_{\mathcal{M}})$ for $\mathcal{M}$ is functional, and we can define a computable partial step function.

▶ **Definition 4** (Partial Step Function, Run)**.** For a machine $\mathcal{M}$, the *partial step function* is given by $\mathcal{M}(x) = y$ if $x \longrightarrow_{\mathcal{M}} y$.

A *run* in $\mathcal{M}$ starting from a configuration $x$ is the (potentially infinite) sequence $x, \mathcal{M}(x), \mathcal{M}^2(x), \mathcal{M}^3(x), \ldots$.

Famously, the halting problem for two-counter machines (Problem 5) is undecidable.

▶ **Problem 5** (Two-Counter Machine Halting)**.** Given a two-counter machine $\mathcal{M}$ and a configuration $x$, is $x$ terminating in $\mathcal{M}$?

Minsky's universality proof for program-machines carries over to CM2, resulting in the undecidability of the corresponding halting problem (Theorem 6). Most importantly, a conditional jump at position $p$ to program index $q$ if counter $c$ is zero can be simulated by the instructions $[\mathtt{dec}_c\,(p+3), \mathtt{inc}_c, \mathtt{dec}_c\, q]$.

▶ **Theorem 6** ([19, Section 11.5] and [19, Theorem 14.1-1])**.** Two-counter machine halting (Problem 5) is undecidable.

▶ **Remark 7.** Theorem 6 is mechanized by Forster et al. [7], as part of the Coq library of undecidability proofs.

## 3 Reversible Machines

In this section we consider *reversible* two-counter machines (also called *backward deterministic*). That is, machines with a right-unique step relation (or, injective step function). We show that for CM2 *reversibility* (is a given machine reversible?) and *reversible halting* (is a given configuration terminating in a given reversible machine?) are decidable problems. This contrasts the negative result by Morita [20, Theorem 4.2] for a different two-counter machine model with a richer instruction set. The positive result is unexpected because CM2 and Morita's machine model can simulate one another preserving determinism (as opposed to backwards-determinism).

▶ **Definition 8** (Reversible Machine)**.** A machine $\mathcal{M}$ is *reversible* if for any configurations $x, y, z$ such that $x \longrightarrow_{\mathcal{M}} z$ and $y \longrightarrow_{\mathcal{M}} z$ we have $x = y$.

▶ **Example 9.** Consider $\mathcal{M} = [\texttt{dec}_0\, 4, \texttt{inc}_0, \texttt{dec}_0\, 0]$ from Example 3. The machine $\mathcal{M}$ is reversible because $(\longrightarrow_{\mathcal{M}})$, fully given below, is right-unique.

$$(0,(0,b)) \xrightarrow{\texttt{dec}_0 4}_{\mathcal{M}} (1,(0,b)) \qquad (0,(a+1,b)) \xrightarrow{\texttt{dec}_0 4}_{\mathcal{M}} (4,(a,b))$$
$$(1,(a,b)) \xrightarrow{\texttt{inc}_0}_{\mathcal{M}} (2,(a+1,b))$$
$$(2,(0,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (3,(0,b)) \qquad (2,(a+1,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (0,(a,b))$$

▶ **Remark 10.** A reversible machine $\mathcal{M}$ cannot contain both instructions $\texttt{dec}_0\, q$ (at position $p_1$) and $\texttt{dec}_1\, q$ (at position $p_2$). Otherwise, the transitions $(p_1,(1,0)) \xrightarrow{\texttt{dec}_0 q}_{\mathcal{M}} (q,(0,0))$ and $(p_2,(0,1)) \xrightarrow{\texttt{dec}_1 q}_{\mathcal{M}} (q,(0,0))$ contradict reversibility of $\mathcal{M}$.

▶ **Remark 11.** Morita gives a different, syntactic definition of reversibility (no *range overlap* [20, Definition 2.3]), specific to the underlying machine model. In fact, Morita's characterization is stronger than right-uniqueness of the step relation (cf. Definition 8), because it also takes into account the instruction used. Therefore, Morita's negative result [20, Theorem 4.2] holds a fortiori, when reversibility is defined by right-uniqueness of the step relation (cf. Definition 3). Compared to the positive result (Theorem 21) in the present work, the richer instruction set is the main contributing factor to undecidability, and not the particular definition of reversibility.

Before we consider the corresponding halting problem, let us ensure that we can decide membership (Corollary 14) in the class of reversible machines.

▶ **Problem 12** (Two-Counter Machine Reversibility)**.** Given a two-counter machine $\mathcal{M}$, is $\mathcal{M}$ reversible?

The following Lemma 13 bounds the set of configurations, which suffices to characterize reversibility.

▶ **Lemma 13.** For a machine $\mathcal{M}$ let $T_{\mathcal{M}} = \{(i,(a,b)) \mid i < |\mathcal{M}|, a \leq 2, b \leq 2\}$. If for all $x, y \in T_{\mathcal{M}}$ such that $\mathcal{M}(x) = \mathcal{M}(y)$ we have $x = y$, then $\mathcal{M}$ is reversible.

**Proof.** Given a machine $\mathcal{M}$, the step relation $(\longrightarrow_{\mathcal{M}})$ only depends on the current program index (bounded by $|\mathcal{M}|$) and on whether the current counters are positive or zero. Additionally, in one step the counters may only increase/decrease by one. Assume for some configurations $x, y, z$ such that $x \neq y$ we have $x \longrightarrow_{\mathcal{M}} z$ and $y \longrightarrow_{\mathcal{M}} z$. By exhaustive case analysis on the instruction taken, we can sufficiently decrease the counter values in $x, y, z$, constructing distinct configurations $x', y' \in T_{\mathcal{M}}$ such that $\mathcal{M}(x') = \mathcal{M}(y')$. ◀

As an immediate consequence of the above Lemma 13, reversibility is decidable (in polynomial time).

▶ **Corollary 14.** Two-counter machine reversibility (Problem 12) is decidable.

Reversible machine halting (Problem 15) is the restriction of the halting problem to reversible machines.

▶ **Problem 15** (Two-Counter Reversible Machine Halting)**.** Given a reversible two-counter machine $\mathcal{M}$ and a configuration $x$, is $x$ terminating in $\mathcal{M}$?

Morita shows that for a two-counter machine model, which is different from CM2, reversible machine halting is undecidable [20, Theorem 4.2]. Surprisingly, this result does not translate to CM2. The main difference is that in Morita's case the richer instruction set allows for reversible nested loops, whereas CM2 does not. In the remainder of this section we give a decision procedure for reversible machine halting for CM2.

The following Example 16 shows that for the reversible machine from Examples 3 and 9 it does not suffice to inspect positivity of the counters in a given configuration to decide termination.

▶ **Example 16.** Consider the reversible machine $\mathcal{M} = [\mathtt{dec}_0\, 4, \mathtt{inc}_0, \mathtt{dec}_0\, 0]$ from Example 3. The configurations $(2,(0,0))$ and $(2,(2,0))$ are terminating, but $(2,(1,0))$ is not, because

$$(2,(0,0)) \xrightarrow{\mathtt{dec}_0 0}_{\mathcal{M}} (3,(0,0)) \text{ halts} \qquad \text{(terminating)}$$
$$(2,(1,0)) \xrightarrow{\mathtt{dec}_0 0}_{\mathcal{M}} (0,(0,0)) \xrightarrow{\mathtt{dec}_0 4}_{\mathcal{M}} (1,(0,0)) \xrightarrow{\mathtt{inc}_0}_{\mathcal{M}} (2,(1,0)) \xrightarrow{\mathtt{dec}_0 0}_{\mathcal{M}} \dots \qquad \text{(non-terminating)}$$
$$(2,(2,0)) \xrightarrow{\mathtt{dec}_0 0}_{\mathcal{M}} (0,(1,0)) \xrightarrow{\mathtt{dec}_0 4}_{\mathcal{M}} (4,(0,0)) \text{ halts} \qquad \text{(terminating)}$$

However, in the above Example 16, considering the program index 0, it *does* suffice to inspect positivity of the counters to decide termination. In fact, this is systematic for reversible CM2, which is the key insight (Lemma 18) to decide the reversible halting problem (Theorem 21). Additionally, any run of a reversible CM2 will eventually halt or reach the program index 0 (Lemma 17).

▶ **Lemma 17.** Let $\mathcal{M}$ be a reversible machine. Given a configuration $x$ we can compute a configuration $(p,(a,b))$ such that $x \longrightarrow^*_{\mathcal{M}} (p,(a,b))$, and $p = 0$ or $p \geq |\mathcal{M}|$.

**Proof.** For $x = (p,(a,b))$ by induction on $\max\{(|\mathcal{M}| - p), 0\}$. The only interesting case is when $\mathtt{dec}_c\, q$ is the $p$-th instruction of $\mathcal{M}$ and $0 < q < |\mathcal{M}|$. In this case, by exhaustive case analysis on the $(q-1)$-th instruction of $|\mathcal{M}|$, we can contradict reversibility of $\mathcal{M}$. ◀

Let us consider the partitioning $\mathcal{T} = \{T_{(0,0)}, T_{(0,1)}, T_{(1,0)}, T_{(1,1)}\}$ of configurations at program index zero with respect to positivity of counters, where

$$T_{(0,0)} = \{(0,(0,0))\}$$
$$T_{(0,1)} = \{(0,(0,b)) \mid 1 \leq b\}$$
$$T_{(1,0)} = \{(0,(a,0)) \mid 1 \leq a\}$$
$$T_{(1,1)} = \{(0,(a,b)) \mid 1 \leq a, 1 \leq b\}$$

The following Lemma 18 shows that for a reversible machine in each partition all configurations exhibit uniform behavior.

▶ **Lemma 18.** Let $\mathcal{M}$ be a reversible machine and let $T \in \mathcal{T}$. We can
1. show for all $x \in T$ that $x$ terminates,
2. show for all $x \in T$ that $x$ does not terminate, or
3. compute $T' \in \mathcal{T}$ such that for all $x \in T$ there is a $y \in T'$ such that $x \longrightarrow^+_{\mathcal{M}} y$.

**Proof.** Let $a_0, b_0 \in \{0,1\}$ and $T_{(a_0,b_0)} \in \mathcal{T}$. We compute the prefix (of length at most $|\mathcal{M}|$) of a run from $(0,(a_0,b_0))$ with increasing program indices. The run either halts or reaches the instruction $\mathtt{dec}_c\, 0$ such that the next configuration is $y = (0,(a',b'))$ (similarly to the proof of Lemma 17). The following case analysis provides an overview over the argument. The individual cases are by case analysis on the possible instructions taken.
**Case $(a_0,b_0) = (0,0)$:** For $T' = T_{(\min\{1,a'\},\min\{1,b'\})}$ we have $(0,(0,0)) \longrightarrow^+_{\mathcal{M}} y \in T'$.

**Case $(a_0, b_0) = (0, 1)$:** There are four cases

**$1 \le a'$ and $1 \le b'$:** for $T' = T_{(1,1)}$ we have $(0, (0, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (a', b+b')) \in T'$.

**$a' = 0$ and $1 \le b'$:** any configuration $(0, (0, b+1))$ is non-terminating.

**$1 \le a'$ and $b' = 0$:** we have $(0, (0, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (a', b))$. By case analysis on $b$, taking Remark 10 into account, and depending on the uniform behavior of configurations in $T_{(1,1)}$, we have that any configuration $(0, (0, b+1))$ is terminating, or for $T' = T_{(1,0)}$ we have that $(0, (0, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (a''+1, 0)) \in T'$ for some $a'' \in \mathbb{N}$.

**$a' = 0$ and $b' = 0$:** for $T' = T_{(0,0)}$ we have $(0, (0, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (0, 0)) \in T'$.

**Case $(a_0, b_0) = (1, 0)$:** Analogous to case $T = T_{(0,1)}$.

**Case $(a_0, b_0) = (1, 1)$:** There are three cases (the case $a' = 0 = b'$ is not possible).

**$1 \le a'$ and $1 \le b'$:** for $T' = T_{(1,1)}$ we have $(0, (a+1, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (a+a', b+b')) \in T'$.

**$a' = 0$ and $1 \le b'$:** for $T' = T_{(0,1)}$ we have $(0, (a+1, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (0, b''+1)) \in T'$ for some $b'' \in \mathbb{N}$.

**$1 \le a'$ and $b' = 0$:** for $T' = T_{(1,0)}$ we have $(0, (a+1, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (a''+1, 0)) \in T'$ for some $a'' \in \mathbb{N}$. ◀

The following Example 19 illustrates the uniform behavior of configurations in each partition in $\mathcal{T}$, as described in the above Lemma 18.

▶ **Example 19.** Consider the machine $\mathcal{M} = [\texttt{inc}_1, \texttt{dec}_0\, 5, \texttt{inc}_0, \texttt{dec}_0\, 0]$. Similarly to Example 9, $\mathcal{M}$ is reversible. The uniform behavior of configurations in each partition in $\mathcal{T}$ is as follows.

- For $(0, (0, 0)) \in T_{(0,0)}$ we have $(0, (0, 0)) \longrightarrow^{+}_{\mathcal{M}} (0, (0, 1)) \in T_{(0,1)}$.
- For all $(0, (0, b+1)) \in T_{(0,1)}$ we have $(0, (0, b+1)) \longrightarrow^{+}_{\mathcal{M}} (0, (0, b+2)) \in T_{(0,1)}$.
- For all $(0, (a+1, 0)) \in T_{(1,0)}$ we have that $(0, (a+1, 0))$ terminates after 2 steps.
- For all $(0, (a+1, b+1)) \in T_{(1,1)}$ we have that $(0, (a+1, b+1))$ terminates after 2 steps.

▶ **Remark 20.** Morita's reversible universal machine construction [20, Proof of Theorem 4.1] requires computational distinction between odd and even counter values. Therefore, for Morita's richer instruction set there is a reversible machine for which the configurations $(0, (2\cdot a, 0))$ are terminating and configurations $(0, (2\cdot a+1, 0))$ are not terminating. In contrast, by Lemma 18 there is no such reversible CM2. For instance, the configurations $(0, (2, 0))$ and $(0, (3, 0))$ are both members of the partition $T_{(1,0)}$, and expose uniform termination behavior for any reversible CM2.

As a result of Lemma 18, termination of configurations with program index 0 is characterized by a finite state automaton with the four states $\mathcal{T}$. Combined with Lemma 17, we obtain a decision procedure for reversible machine halting (Theorem 21).

▶ **Theorem 21.** Two-counter reversible machine halting (Problem 15) is decidable.

**Proof.** Given a reversible machine $\mathcal{M}$ and a configuration $x$, by Lemma 17 compute the configuration $(p, (a, b))$ such that $x \longrightarrow^{*}_{\mathcal{M}} (p, (a, b))$ and $p = 0$ or $p \ge |\mathcal{M}|$. In case $p \ge |\mathcal{M}|$ we have that $(p, (a, b))$ halts, and therefore $x$ is terminating. If $p = 0$, then for $a_0 = \min\{1, a\}$ and $b_0 = \min\{1, b\}$ we have $(p, (a, b)) \in T_{(a_0,b_0)}$. Using Lemma 18, compute the finite state automaton with states $\mathcal{T}$, where the initial state is $T_{(a_0,b_0)}$, the accepting states satisfy (18.1), and the transition function corresponds to (18.3). Termination of $x \in T_{(a_0,b_0)}$ corresponds to reachability of any accepting state in the constructed finite automaton, which is decidable. ◀

▶ **Remark 22.** The proof of Theorem 21 entails a *polynomial time* decision procedure for reversible halting. Notably, in order to construct the finite state automaton with states $\mathcal{T}$ for a given machine $\mathcal{M}$, it suffices to inspect a constant number of runs of length at most $|\mathcal{M}|$.

# 4 Boundedness

In this section we consider boundedness properties of two-counter machines. First, we recall that *total boundedness* (does every run eventually halt or enter a configuration cycle?) is undecidable [14, Theorem 8], which also holds for the machine model at hand (Theorem 28). Second, for *uniform boundedness* (is there a uniform bound on the number of reachable configurations?) we contribute a decision procedure (Theorem 40). Techniques presented in this section are not specific to CM2 and extend to other two-counter machine models.

If a run starting from a configuration $x$ in a machine $\mathcal{M}$ halts or enters a configuration cycle, then we can bound the number of reachable configurations from $x$ in $\mathcal{M}$ (Definition 23).

▶ **Definition 23** (Bound)**.** An $n \in \mathbb{N}$ *bounds* a configuration $x$ in a machine $\mathcal{M}$ if we have $|\{y \mid x \longrightarrow^*_{\mathcal{M}} y\}| \leq n$.

For a *totally bounded* machine every configuration is bounded. In other words, a totally bounded machine has no aperiodic, non-terminating runs.

▶ **Definition 24** (Totally Bounded Machine)**.** A machine $\mathcal{M}$ is *totally bounded* if for all configurations $x$ there exists an $n \in \mathbb{N}$ such that $n$ bounds $x$ in $\mathcal{M}$.

▶ **Remark 25.** Every (possibly infinite) run of a totally bounded machine can be fully described by a finite prefix. This renders key properties such as reachability and termination decidable, and is useful for model-checking.

▶ **Example 26.** Consider $\mathcal{M} = [\texttt{dec}_0\,0, \texttt{inc}_0, \texttt{dec}_0\,0]$. Configurations $(0, (a, b))$ are bounded by $a + 3$ in $\mathcal{M}$ because

$$(0,(a,b)) \longrightarrow^a_{\mathcal{M}} (0,(0,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (1,(0,b)) \xrightarrow{\texttt{inc}_0}_{\mathcal{M}} (2,(1,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (0,(0,b)) \longrightarrow_{\mathcal{M}} \dots$$

Overall, the machine $\mathcal{M}$ is totally bounded with the following bounds

| | |
|---|---|
| $(0,(a,b))$ | bounded by $a+3$ |
| $(1,(a,b)) \xrightarrow{\texttt{inc}_0}_{\mathcal{M}} (2,(a+1,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (0,(a,b))$ | bounded by $a+5$ |
| $(2,(0,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (3,(0,b))$ halts | bounded by 2 |
| $(2,(a+1,b)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (0,(a,b))$ | bounded by $a+4$ |
| $(p+3,(a,b))$ halts | bounded by 1 |

The negative result by Kuzmin and Chalyy [14, Theorem 8] for two-counter machine total boundedness (Problem 27) translates to the machine model at hand (Theorem 28).

▶ **Problem 27** (Two-Counter Machine Total Boundedness)**.** Given a two-counter machine $\mathcal{M}$, is $\mathcal{M}$ totally bounded?

▶ **Theorem 28** ([14, Theorem 8])**.** Two-counter machine total boundedness (Problem 27) is undecidable.

In contrast to a totally bounded machine, for a *uniformly bounded* machine the bound on the number of reachable configurations does not depend on the starting configuration.

▶ **Definition 29** (Uniform Bound)**.** An $n \in \mathbb{N}$ *uniformly bounds* a machine $\mathcal{M}$ if $n$ bounds all configurations $x$ in $\mathcal{M}$.

▶ **Definition 30** (Uniformly Bounded Machine). A machine $\mathcal{M}$ is *uniformly bounded* if there exists a uniform bound $n$ of $\mathcal{M}$.

Intuitively, a uniformly bounded machine operates in bounded space, given by the particular uniform bound. The following Example 31 shows that uniform boundedness is strictly stronger than total boundedness.

▶ **Example 31.** Consider the totally bounded machine $\mathcal{M} = [\texttt{dec}_0\,0, \texttt{inc}_0, \texttt{dec}_0\,0]$ from Example 26. There is no uniform bound for $\mathcal{M}$ because for any $n \in \mathbb{N}$ from the configuration $(0, (n, 0))$ more than $n$ distinct configurations are reachable.

▶ **Remark 32.** It is surprising that any model of computation can have computationally interesting, uniformly bounded machines. For example, such machines are not capable of processing an arbitrarily large input. However, for Turing machines, relying on the ingenious technique developed by Hooper [10] one can reduce Turing machine halting to a uniform boundedness problem for stack machines [2].

▶ **Problem 33** (Two-Counter Machine Uniform Boundedness). Given a two-counter machine $\mathcal{M}$, is $\mathcal{M}$ uniformly bounded?

In the remainder of this section we give a decision procedure for two-counter machine uniform boundedness.

First, we characterize whether $n$ bounds a configuration $x$ by inspection of at most the first $n$ steps in a run from $x$.

▶ **Fact 34.** An $n \in \mathbb{N}$ bounds a configuration $x$ in a machine $\mathcal{M}$ iff either $\mathcal{M}^n(x)$ is undefined or $\mathcal{M}^n(x) = \mathcal{M}^m(x)$ for some $m < n$.

The above Fact 34 entails a decision procedure to determine whether $n$ bounds $x$ in $\mathcal{M}$.

▶ **Corollary 35.** Given a machine $\mathcal{M}$, a configuration $x$, and an $n \in \mathbb{N}$, it is decidable whether $n$ bounds $x$ in $\mathcal{M}$.

Second, we characterize whether $n$ bounds a machine $\mathcal{M}$ by inspection of the finitely many configurations with counters at most $n$.

▶ **Lemma 36.** Let $\mathcal{M}$ be a machine and let $n \in \mathbb{N}$. If for all $p \leq |\mathcal{M}|$, $a \leq n$, and $b \leq n$ we have that $n$ bounds $(p, (a, b))$ in $\mathcal{M}$, then $n$ uniformly bounds $\mathcal{M}$.

**Proof.** By Fact 34, for any configuration $x$ it suffices to inspect the first $n$ steps of any run from $x$ to decide whether $n$ bounds $x$. Since at each step the counter values change by at most one, configurations with counter values of at least $n$ behave uniformly after at most $n$ steps with respect to halting or entering a configuration cycle. ◀

As a result of the above Lemma 36 and Corollary 35, it is decidable whether $n$ bounds $\mathcal{M}$.

▶ **Corollary 37.** Given a machine $\mathcal{M}$ and an $n \in \mathbb{N}$, it is decidable whether $n$ uniformly bounds $\mathcal{M}$.

Third, we give a sufficient condition for aperiodic, arbitrary long runs (Lemma 38). A machine satisfying this condition cannot be uniformly bounded. Intuitively, the condition captures repeatable program index cycles for which at least one counter value changes. Considering counter values which are at least the cycle length, larger counter values exhibit identical control flow.

▶ **Lemma 38.** Let $\mathcal{M}$ be a machine, let $k \in \mathbb{N}$, and let $(p,(a_1,b_1)),(p,(a_2,b_2))$ be configurations such that $(p,(a_1,b_1)) \longrightarrow^k_{\mathcal{M}} (p,(a_2,b_2))$. If all of the following conditions hold, then $\mathcal{M}$ is not uniformly bounded.

(1) $k \le a_1$ or $a_1 = a_2$ (2) $k \le b_1$ or $b_1 = b_2$ (3) $a_1 \ne a_2$ or $b_1 \ne b_2$

**Proof.** Assume that some $n$ uniformly bounds $\mathcal{M}$. Consider the case $k \le a_1 \ne a_2$ and $b_1 = b_2 = b$ (the other cases are analogous). In the first $k-1$ steps from $(p,(a_1,b_1))$ the first counter is positive. Therefore, from the configuration $(p,(a_1+n\cdot a_1,b))$ there are more than $n$ distinct reachable configurations:

$$\begin{aligned}(p,(a_1+n\cdot a_1,b)) &\longrightarrow^k_{\mathcal{M}} (p,(a_1+(n-1)\cdot a_1+1\cdot a_2,b))\\ &\longrightarrow^k_{\mathcal{M}} (p,(a_1+(n-2)\cdot a_1+2\cdot a_2,b))\\ &\longrightarrow^k_{\mathcal{M}} \dots \longrightarrow^k_{\mathcal{M}} (p,(a_1+n\cdot a_2,b))\end{aligned}$$

This contradicts the uniform bound $n$ of $\mathcal{M}$. ◀

Fourth, we characterize whether $\mathcal{M}$ is uniformly bounded by inspection of the particular bound $(|\mathcal{M}|+1)^5$. By the pigeonhole principle this upper bound covers sufficiently many configurations to exploit the negative condition (Lemma 38) for uniform boundedness.

▶ **Lemma 39.** If a machine $\mathcal{M}$ is uniformly bounded, then $(|\mathcal{M}|+1)^5$ uniformly bounds $\mathcal{M}$.

**Proof.** For an arbitrary configuration $x$, consider the first at most $(|\mathcal{M}|+1)^5$ steps of a run in $\mathcal{M}$ from $x$. If the run halts or enters a configuration cycle, then $(|\mathcal{M}|+1)^5$ bounds $x$ in $\mathcal{M}$. Otherwise, we contradict that $\mathcal{M}$ is uniformly bounded as follows. In this case, the considered $(|\mathcal{M}|+1)^5$ configurations are distinct (and defined).

Let $l = |\mathcal{M}|\cdot(|\mathcal{M}|+1)$. By the pigeonhole principle, in the first $|\mathcal{M}|\cdot l^2$ steps, we encounter a configuration $x_1 = (p_1,(a_1,b_1))$ such that $l \le a_1$ or $l \le b_1$. Consider $l \le a_1$ (the other case is analogous). For the the next $|\mathcal{M}|^2$ steps the first counter is at least $|\mathcal{M}|$, and there are two cases.

**Case 1:** We encounter a configuration $x_2 = (p_2,(a_2,b_2))$ such that $|\mathcal{M}| \le a_2$ and $|\mathcal{M}| \le b_2$. By the pigeonhole principle, in the next $|\mathcal{M}|$ steps, we necessarily encounter configurations $x_3 = (p,(a_3,b_3))$ and $x_4 = (p,(a_4,b_4))$ such that $x_3 \longrightarrow^k_{\mathcal{M}} x_4$, $k \le a_3$, $k \le b_3$, and $x_3 \ne x_4$. This contradicts uniform boundedness of $\mathcal{M}$ by Lemma 38.

**Case 2:** All configurations are such that the second counter is less than $|\mathcal{M}|$. By the pigeonhole principle, we encounter configurations $x'_3 = (p',(a'_3,b'))$ and $x'_4 = (p',(a'_4,b'))$ such that $x'_3 \longrightarrow^k_{\mathcal{M}} x'_4$ and $k \le a'_3 \ne a'_4$. This contradicts uniform boundedness of $\mathcal{M}$ by Lemma 38. ◀

Finally, relying on the above Lemma 39 and Corollary 37, we give a decision procedure for uniform boundedness.

▶ **Theorem 40.** Two-counter machine uniform boundedness (Problem 27) is decidable.

**Proof.** Given a machine $\mathcal{M}$, by Lemma 39 it suffices to decide whether $(|\mathcal{M}|+1)^5$ uniformly bounds $\mathcal{M}$, which is decidable by Corollary 37. ◀

▶ **Remark 41.** The proof of Theorem 40 entails a *polynomial time* decision procedure for uniform boundedness. In particular, given a machine $\mathcal{M}$ we inspect the potential uniform bound $n = (|\mathcal{M}|+1)^5$ (cf. Lemma 39). That is, we inspect whether $n$ bounds configurations $(p,(a,b))$ such that $p \le |\mathcal{M}|$, $a \le n$, and $b \le n$ (cf. Lemma 36). Each of these $|\mathcal{M}|\cdot n^2$ configurations halts or enters a configuration cycle in the first $n$ steps in $\mathcal{M}$ (cf. Fact 34) iff $\mathcal{M}$ is uniformly bounded.

## 5 Mortality

In this section we consider mortality properties of two-counter machines. First, we recall that *total mortality* (does every run eventually halt?) is undecidable [10, Part VI.7], which also holds for the machine model at hand (Theorem 45). Second, for *uniform mortality* (is there a uniform bound on the number of steps from any configuration?) we give a decision procedure (Theorem 49).

▶ **Definition 42** (Totally Mortal Machine)**.** A machine $\mathcal{M}$ is *totally mortal* if all configurations $x$ are terminating in $\mathcal{M}$.

Total mortality is strictly stronger than total boundedness (Example 43).

▶ **Example 43.** Consider the totally bounded machine $\mathcal{M} = [\texttt{dec}_0\, 0, \texttt{inc}_0, \texttt{dec}_0\, 0]$ from Example 26. The machine $\mathcal{M}$ is not totally mortal because of the non-terminating run

$$(0,(0,0)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (1,(0,0)) \xrightarrow{\texttt{inc}_0}_{\mathcal{M}} (2,(1,0)) \xrightarrow{\texttt{dec}_0 0}_{\mathcal{M}} (0,(0,0)) \longrightarrow_{\mathcal{M}} \dots$$

The original negative result by Hooper [10, Part VI.7] for two-counter machine total mortality (Problem 44) translates to the machine model at hand (Theorem 45).

▶ **Problem 44** (Two-Counter Machine Total Mortality)**.** Given a two-counter machine $\mathcal{M}$, is $\mathcal{M}$ totally mortal?

▶ **Theorem 45** ([10, Part VI.7])**.** Two-counter machine total mortality (Problem 44) is undecidable.

▶ **Remark 46.** The negative result for *reversible* two-counter machine total mortality [12, Theorem 1] does *not* hold for CM2. By Lemma 18, it suffices to inspect reachability of accepting states in a computable finite state automaton, which is decidable.

In a *uniformly mortal* machine there is a uniform bound on the number of steps after which every run halts.

▶ **Definition 47** (Uniformly Mortal Machine)**.** A machine $\mathcal{M}$ is *uniformly mortal* if there exists an $n \in \mathbb{N}$ such that for any configuration $x$ we have that $\mathcal{M}^n(x)$ is undefined.

Kari and Ollinger sketch a decision procedure [12, Theorem 2] for uniform mortality (Problem 48). In the remainder of this section we give an alternative decision procedure (Theorem 49), based on the decision procedure for uniform boundedness.

▶ **Problem 48** (Two-Counter Machine Uniform Mortality)**.** Given a two-counter machine $\mathcal{M}$, is $\mathcal{M}$ uniformly mortal?

▶ **Theorem 49.** Two-counter machine uniform mortality (Problem 48) is decidable.

**Proof.** Given a machine $\mathcal{M}$, decide whether $\mathcal{M}$ is uniformly bounded. If not, then $\mathcal{M}$ is not uniformly mortal. Otherwise, $n = (|\mathcal{M}| + 1)^5$ uniformly bounds $\mathcal{M}$ by Lemma 39. It suffices to decide whether $n$ bounds the maximal number of steps from any configuration. Configurations with counter values at least $n$ behave uniformly for the first $n$ steps. Therefore, it suffices to inspect the finitely many configurations $(p,(a,b))$ such that $p \le |\mathcal{M}|$, $a \le n$, $b \le n$. All such configurations halt after at most $n$ steps iff $\mathcal{M}$ is uniformly mortal. ◀

▶ **Remark 50.** The proof of Theorem 49 entails a *polynomial time* decision procedure for uniform mortality. In particular, given a machine $\mathcal{M}$ we inspect whether $n = (|\mathcal{M}| + 1)^5$ bounds the number of steps from configurations $(p,(a,b))$ such that $p \le |\mathcal{M}|$, $a \le n$, and $b \le n$. Each of these $|\mathcal{M}| \cdot n^2$ configurations halts in at most $n$ steps iff $\mathcal{M}$ is uniformly mortal.

# 6 Mechanization

At the level of human intuition, the decidability results for reversible halting (Theorem 21), uniform boundedness (Theorem 40), and uniform mortality (Theorem 49) are uncomplicated. However, the necessary verification of the individual results in full-detail, often by nested case analysis, is laborious and (without computer assistance) error-prone.

A mechanization of the presented results using a proof assistant constitutes a rigorous, mechanically verifiable correctness proof. Using the Coq proof assistant [27] in particular has several benefits. First, as argued by Forster [3, Chapter 2], Coq is well-suited for positive and negative computability results because of its separate impredicative universe of propositions, besides a computational type hierarchy. This separation allows for a distinction between computational and non-computational aspects of computability results. In addition, any function implemented in axiom-free Coq is, by design, total and computable. Second, the Coq library of undecidability proofs [8] is a readily available uniform framework to mechanize computability results. The present work heavily relies on the existing infrastructure for two-counter machines provided by the library. Third, using the `Extraction` framework [17] one can extract effective implementations of the individual decision procedures.

The library defines[3] decidability of decision problems as follows.

```
Definition reflects (b : bool) (p : Prop) := p <-> b = true.

Definition decider {X} (f : X -> bool) (P : X -> Prop) : Prop :=
  forall x, reflects (f x) (P x).

Definition decidable {X} (P : X -> Prop) : Prop :=
  exists f : X -> bool, decider f P.
```

In particular, a problem `P : X -> Prop` on the domain `X` is decidable, if there exists a Boolean function `f : X -> bool` such that for all `x` in `X` we have that `P x` holds iff `f x = true` (cf. *small scale reflection* [9]).

In general, a propositional existence proof of a computable decision procedure can rely on non-constructive principles, such as the principle of excluded middle. In the present work we mechanize the individual decision procedures in axiom-free Coq, which constitutes the strongest result with respect to constructive mathematics.

The library contains[4] the following definition of two-counter machines (cf. Definition 2).

```
Definition Config : Set := nat * (nat * nat).

Definition state (x: Config) : nat := fst x.
Definition value1 (x: Config) : nat := fst (snd x).
Definition value2 (x: Config) : nat := snd (snd x).

Inductive Instruction : Set :=
  | inc : bool -> Instruction
  | dec : bool -> nat -> Instruction.

Definition Cm2 : Set := list Instruction.

Definition step (M: Cm2) (x: Config) : option Config :=
  [...]

Definition steps (M: Cm2) (k: nat) (x: Config) : option Config :=
  Nat.iter k (obind (step M)) (Some x).
```

[3] `theories/Synthetic/Definitions.v`
[4] `theories/CounterMachines/CM2.v`

In the above, a two-counter machine of type `Cm2` with the configuration space `nat * (nat * nat)` is a list of instructions of shape either `(inc false)` for $\mathtt{inc}_0$, `(inc true)` for $\mathtt{inc}_1$, `(dec false q)` for $\mathtt{dec}_0\, q$, or `(dec true q)` for $\mathtt{dec}_1\, q$. The function `step : Cm2 -> Config -> option Config`, mechanizes the two-counter machine partial step function (Definition 4). For exactly the halting configurations `x : Config` we have `step M x = None`. The iterated step function is `steps : Cm2 -> nat -> Config -> option Config`. A prominent result[5] in the library is the undecidability of the halting problem for two-counter machines.

The remainder of this section outlines the mechanization of the decision procedures contributed by the present work to the Coq library of undecidability proofs.

## Reversible Halting

The following predicate `CM2_REV_HALT` mechanizes the reversible halting problem for two-counter machines (Problem 15).

```
Definition terminating (M: Cm2) (x: Config) :=
  exists k, steps M k x = None.

Definition reversible (M : Cm2) : Prop :=
  forall x y z, step M x = Some z -> step M y = Some z -> x = y.

Definition CM2_REV_HALT : { M: Cm2 | reversible M } * Config -> Prop :=
  fun '((exist _ M _), x) => terminating M x.
```

In particular, given[6] a two-counter machine `M : Cm2`, a proof that `M` is `reversible` (`step` is injective), and a configuration `x : Config`, is there a `k : nat` such that `M` halts after at most `k` steps starting from configuration `x`?

The decision procedure `decide : { M: Cm2 | reversible M } * Config -> bool` for the predicate `CM2_REV_HALT` is mechanized in `theories/CounterMachines/Deciders/CM2_REV_HALT_dec.v` with the corresponding correctness proof `decide_spec : decider decide CM2_REV_HALT`. Notably, the key Lemma 18 for the construction of a finite state automaton to decide reversible halting is mechanized as follows (`RZ` mechanizes membership in the same partition in $\mathcal{T}$).

```
Lemma uniform_transition ab :
  In ab representatives ->
  (forall a'b', RZ ab a'b' -> terminating (0, a'b')) +
  (forall a'b', RZ ab a'b' -> non_terminating (0, a'b')) +
  (* uniform transition *)
  {v | In v representatives /\
    (forall a'b', RZ ab a'b' -> exists w, RZ v w /\
      reaches_plus (0, a'b') (0, w)) }.
```

In order to implement a computable decision procedure, it is important to use computational disjunction (`+`) and the dependent pair `{ v | In v representatives /\ ... }`. The corresponding propositional counterparts (`\/`) and (`exists v, In v representatives /\ ...`) do not suffice.

The overall mechanization spans approximately 1000 LOC. It relies heavily on the proof automation tactic `lia` for linear integer arithmetic.

[5] `theories/CounterMachines/CM2_undec.v`
[6] The syntax `'((exist _ M _), x)` matches a member of `{ M: Cm2 | reversible M } * Config`, and binds the given machine `M : Cm2` and the given configuration `x : Config`.

## Uniform Boundedness

The following predicate `CM2_UBOUNDED` mechanizes uniform boundedness for two-counter machines (Problem 33).

```
Definition reaches (M: Cm2) (x y: Config) :=
  exists k, steps M k x = Some y.

Definition bounded (M: Cm2) (k: nat) (x: Config) : Prop :=
  exists (L: list Config), (length L <= k) /\
    (forall (y: Config), reaches M x y -> In y L).

Definition uniformly_bounded (M: Cm2) : Prop :=
  exists k, forall x, bounded M k x.

Definition CM2_UBOUNDED : Cm2 -> Prop :=
  fun M => uniformly_bounded M.
```

In particular, given a two-counter machine `M : Cm2`, is there a bound `k : nat` such that for all configurations `x : Config` the reachable configurations from `x` are bounded by a list `L : list Config` of length at most `k`?

The decision procedure `decide : Cm2 -> bool` for the predicate `CM2_UBOUNDED` is mechanized in `theories/CounterMachines/Deciders/CM2_UBOUNDED_dec.v` with the corresponding correctness proof `decide_spec : decider decide CM2_UBOUNDED`. Notably, the key Lemma 39 which provides an uniform upper bound is mechanized as follows (where `l` is `length M`).

```
Lemma bound_on_uniform_bound : uniformly_bounded M ->
  forall x, bounded M ((l+1)*(l+1)*(l+1)*(l+1)*(l+1)) x.
```

The overall mechanization spans approximately 400 LOC. It relies on the following negative pigeonhole principle.

```
Lemma pigeonhole {X : Type} (L L' : list X) :
  incl L L' -> length L' < length L -> not (NoDup L).
```

In particular, given two lists `L` and `L'`, if each element of `L` is in the strictly shorter list `L'`, then `L` is not duplicate-free.

## Uniform Mortality

The following predicate `CM2_UMORTAL` mechanizes uniform mortality for two-counter machines (Problem 48).

```
Definition mortal (M: Cm2) (k: nat) (x: Config) : Prop :=
  steps M k x = None.

Definition uniformly_mortal (M: Cm2) : Prop :=
  exists k, forall x, mortal M k x.

Definition CM2_UMORTAL : Cm2 -> Prop :=
  fun M => uniformly_mortal M.
```

In particular, given a two-counter machine `M : Cm2`, is there a bound `k : nat` such that for all configurations `x : Config` the machine `M` starting from `x` halts after at most `k` steps?

The decision procedure `decide : Cm2 -> bool` for the predicate `CM2_UMORTAL` is mechanized in `theories/CounterMachines/Deciders/CM2_UMORTAL_dec.v` together with the corresponding correctness proof `decide_spec : decider decide CM2_UMORTAL`. The mechanization spans approximately 100 LOC and relies on the previously described mechanized decision procedure for uniform boundedness.

### Extraction

Using the `Extraction` framework [17], effective implementations of the individual decision procedures (in the OCaml programming language) can be obtained from the provided mechanization. For example, the following code mechanizes Example 9 and Example 16.

```
From Undecidability Require Import CM2_REV_HALT_dec.
From Coq Require Import List Extraction.
Import CM2 ListNotations.

Definition M := [dec false 4; inc false; dec false 0].

Lemma HM : reversible M.
Proof.
  intros [[|[|[|[|xp]]]] [[|xa] xb]] [[|[|[|[|yp]]]] [[|ya] yb]] z.
  all: now cbn; congruence.
Qed.

Definition configs := [(2, (0, 0)); (2, (1, 0)); (2, (2, 0))].

Definition results := map (fun x => decide (exist _ M HM, x)) configs.
```

In the above, `M` mechanizes the machine $[\texttt{dec}_0\, 4, \texttt{inc}_0, \texttt{dec}_0\, 0]$ from Example 9 and `HM` certifies reversibility of `M` by automated case analysis. Notably, the proof automation tactic `congruence` (implementing a congruence closure algorithm [23]) is well-suited for automated injectivity proofs. The list `configs` contains the three starting configurations $(2,(0,0))$, $(2,(1,0))$, and $(2,(2,0))$ from Example 16, and the list `results` contains the corresponding halting decisions. Finally, using the command "`Recursive Extraction results.`" an OCaml implementation can be extracted. Upon execution, `results` returns the answers `true` for termination of the configurations $(2,(0,0))$ and $(2,(2,0))$, and `false` for termination of the configuration $(2,(1,0))$, in agreement with Example 16. While still a toy example, this highlights both the suitability of the Coq proof assistant for (practical) computability theory, and the maturity of the underlying tool chain. Additionally, extraction to a widely-used programming language comes with toolchain, performance, and integration benefits for users outside of the proof assistant community.

## 7 Conclusion

The present work gives certified decision procedures for reversible halting (Theorem 21), uniform boundedness (Theorem 40), and uniform mortality (Theorem 49) for CM2, which is an established, computationally universal notion of two-counter machines.

The positive result for reversible halting contrasts universality of reversible two-counter machines [20] with a different, richer instruction set. The presented argument is by modeling relevant control flow of reversible CM2 by a finite state automaton. This renders the established instruction set of CM2 not computationally universal in a reversible setting.

The presented positive results for uniform boundedness (cf. [1, Remark 28]) and uniform mortality (cf. [12, Theorem 2]) provide insight into algorithmic complexity of the respective decision problems. In particular, the underlying arguments are based on a polynomial upper bound on the size of the relevant configuration space.

The described decision procedures are implemented and verified using the Coq proof assistant. Coq is well-suited for positive and negative computability results [3], which in practice culminates in a growing Coq library containing such results. By design, any function implemented in axiom-free Coq is computable and is equipped with a termination certificate. Therefore, both negative results (via computable reduction functions) and positive results

(via computable decision functions) can be presented and verified in a uniform framework. As added benefit, the complementary positive and negative results for individual problem classes (such as problems for two-counter machines) can rely on common infrastructure, avoiding code duplication. The library is well-maintained, which increases longevity of the contributed mechanization.

While the present work focuses on the positive results for CM2, it is desirable for the library to include mechanizations of the known negative results for total boundedness and total mortality.

Unfortunately, in contrast to computability, it is challenging to reason about time or space complexity of Coq code (cf. the active line of work by Kunze and Forster [4, 5, 6]). Therefore, certification of the polynomial time complexity (Remarks 22, 41, and 50) of the decision procedures given in the present work remains open.

The OCaml implementation of the individual decision procedures given by the `Extraction` framework is effective for the toy examples in the present work. However, it is neither efficient nor humanly readable due to the use of `ssreflect` proof tactic language [9]. This can be addressed by a more strict separation between decision procedures, termination proofs, and correctness proofs (cf. the *Braga method* [16] and the `Equations` framework [25]).

## References


1. 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, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSCD.2020.9.
2. Andrej Dudenhefner. Constructive many-one reduction from the halting problem to semi-unification. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, volume 216 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2022.18.
3. Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021.
4. Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from Coq to call-by-value $\lambda$-calculus. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITP.2019.17.
5. Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value $\lambda$-calculus is reasonable for both time and space. *Proc. ACM Program. Lang.*, 4(POPL):27:1–27:23, 2020. doi:10.1145/3371095.
6. Yannick Forster, Fabian Kunze, Gert Smolka, and Maximilian Wuttke. A mechanised proof of the time invariance thesis for the weak call-by-value $\lambda$-calculus. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.19.
7. 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.

8 Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq Library of Undecidable Problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: `https://github.com/uds-psl/coq-library-undecidability`.

9 Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J. Formaliz. Reason.*, 3(2):95–152, 2010. `doi:10.6092/issn.1972-5787/1979`.

10 Philip K. Hooper. The undecidability of the Turing machine immortality problem. *J. Symb. Log.*, 31(2):219–234, 1966. `doi:10.2307/2269811`.

11 Sergiu Ivanov, Elisabeth Pelz, and Sergey Verlan. Small universal non-deterministic Petri nets with inhibitor arcs. In Helmut Jürgensen, Juhani Karhumäki, and Alexander Okhotin, editors, *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, volume 8614 of *Lecture Notes in Computer Science*, pages 186–197. Springer, 2014. `doi:10.1007/978-3-319-09704-6_17`.

12 Jarkko Kari and Nicolas Ollinger. Periodicity and immortality in reversible computing. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008, 33rd International Symposium, MFCS 2008, Torun, Poland, August 25-29, 2008, Proceedings*, volume 5162 of *Lecture Notes in Computer Science*, pages 419–430. Springer, 2008. `doi:10.1007/978-3-540-85238-4_34`.

13 Ivan Korec. Small universal register machines. *Theor. Comput. Sci.*, 168(2):267–301, 1996. `doi:10.1016/S0304-3975(96)00080-1`.

14 Egor Kuzmin and Dmitry Chalyy. Decidability of boundedness problems for Minsky counter machines. *Autom. Control. Comput. Sci.*, 44(7):387–397, 2010. `doi:10.3103/S0146411610070047`.

15 Dominique Larchey-Wendling and Yannick Forster. Hilbert's tenth problem in Coq. 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 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.FSCD.2019.27`.

16 Dominique Larchey-Wendling and Jean-François Monin. Simulating induction-recursion for partial algorithms. In *24th International Conference on Types for Proofs and Programs, TYPES 2018*, 2018.

17 Pierre Letouzey. A new Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002. `doi:10.1007/3-540-39185-1_12`.

18 Marvin Minsky. Recursive unsolvability of Post's problem of "tag" and other topics in theory of Turing machines. *Annals of Mathematics*, pages 437–455, 1961.

19 Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

20 Kenichi Morita. Universality of a reversible two-counter machine. *Theor. Comput. Sci.*, 168(2):303–320, 1996. `doi:10.1016/S0304-3975(96)00081-3`.

21 Kenichi Morita. Reversible computing systems, logic circuits, and cellular automata. In *Third International Conference on Networking and Computing, ICNC 2012, Okinawa, Japan, December 5-7, 2012*, pages 1–8. IEEE Computer Society, 2012. `doi:10.1109/ICNC.2012.10`.

22 Kenichi Morita. *Theory of Reversible Computing*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2017. `doi:10.1007/978-4-431-56606-9`.

23 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`.

24 Emanuele Rodaro and Pedro V. Silva. Amalgams of inverse semigroups and reversible two-counter machines. *Journal of Pure and Applied Algebra*, 217(4):585–597, 2013.

25 Matthieu Sozeau and Cyprien Mangin. Equations v1.2, May 2019. `doi:10.5281/zenodo.3012649`.

26 Simon Spies and Yannick Forster. Undecidability of higher-order unification formalised in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 143–157. ACM, 2020. doi:10.1145/3372885.3373832.

27 The Coq Development Team. The Coq proof assistant, January 2022. doi:10.5281/zenodo.5846982.

# Strategies for Asymptotic Normalization

**Claudia Faggian**
IRIF, CNRS, Université de Paris Cité, F-75013 Paris, France

**Giulio Guerrieri**
Huawei Research, Edinburgh Research Centre, UK

## Abstract

We present an abstract technique to study normalizing strategies when termination is asymptotic, that is, it appears as a limit. Asymptotic termination occurs in several settings, such as effectful, and in particular probabilistic computation – where the limits are distributions over the possible outputs – or infinitary lambda-calculi – where the limits are infinitary terms such as Böhm trees.

As a concrete application, we obtain a result which is of independent interest: a normalization theorem for Call-by-Value (and – in a uniform way – for Call-by-Name) probabilistic lambda-calculus.

**2012 ACM Subject Classification** Theory of computation → Models of computation; Theory of computation → Equational logic and rewriting; Theory of computation → Lambda calculus

**Keywords and phrases** rewriting, strategies, normalization, lambda calculus, probabilistic rewriting

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.17

**Related Version** *Extended Version*: `http://arxiv.org/abs/2204.08772`

**Funding** Work supported by the ANR project PPS: ANR-19-CE48-0014.

## 1 Introduction

Probabilistic computation is an example of computational paradigm where the notion of termination is *asymptotic*, that is, it appears as a *limit*, as opposed to reaching a normal form in a *finite* number of steps. Streams, infinitary $\lambda$-calculus, algebraic rewriting systems, effectful computation, are other examples: the notion of asymptotic computation is pervasive. Here, we investigate asymptotic normalization, and propose a technique to prove that a strategy is guaranteed to produce a maximal or – ideally – the best possible result. Our technique is abstract (in the sense of Abstract Rewriting Systems) and so of general application.

Rewriting is a foundation for the operational theory of formal calculi and programming languages – $\lambda$-calculus being the paradigmatic example where rewriting is an abstract form of program execution. Even if a programming language is usually defined by a specific *evaluation strategy*, to have a general rewriting theory allows for *program transformations, optimizations, parallel/distributed implementations*, and provides a base on which to reason about program equivalence. The $\lambda$-calculus has a rich theory that studies the properties of reductions. Asymptotic computation is much less understood from a rewriting point of view, with the notable exception of infinitary $\lambda$-calculus, whose rewriting theory, pioneered in [9, 22, 23], has been extensively studied.

The process of rewriting describes the *computation of a result.* Normal forms, head normal forms, values, may or must termination, are all possible notions of result. For concreteness, let us focus on normal forms. Operationally, key questions about a system are the existence and uniqueness of normal forms, but also *how* the result is computed. In a *finitary setting* we would ask: may a computation produce a result (*Existence* of normal forms)? If so, is the result unique? Do different computations on the same input lead to the same result (*Uniqueness* of normal forms)? How to compute a result? Is there a reduction strategy that is guaranteed to output a result, if any exists (*Normalizing strategy*)? In the *asymptotic case*, such questions are still relevant, but need to be opportunely formulated. To answer

© Claudia Faggian and Giulio Guerrieri;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 17; pp. 17:1–17:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

them, we need suitable tools and techniques, because those for finitary computation do not necessarily transfer (the key game-changer being that asymptotic termination does not provide a well-founded order, see [13] for examples in a probabilistic setting).

**Abstract Asymptotic Rewriting.** Our approach is to study asymptotic reduction strategies and properties of limits in an *abstract* way (independent of the specific syntax of a calculus) as the theory of Abstract Rewrite Systems (ARS) does for finitary computation, so to isolate proof-techniques which are of general application. For example, in infinitary lambda calculus, the limit is usually a (possibly infinite) limit term, while in probabilistic lambda calculus, the limit is a distribution over (finite) terms. The former is concerned with the depth of the redexes, the latter with the probability of reaching a result. The abstract notions of limit and normalization subsume both, and so abstract results apply to either setting. A further, conceptual advantage of an abstract approach is to display the essence of the arguments, an to neatly discriminate between those properties that rely on specific structure of a concrete setting, and those that belong to any asymptotic notion of computation.

Specifically, we work in the setting of *Quantitative Abstract Rewrite System* (QARS) [14], a framework to study asymptotic rewriting *abstractly* which refines Ariola and Blom's ARSI [4].

**From normal forms to limits.** Intuitively, a possibly infinite reduction sequence $\langle t_n \rangle_n$ from $t = t_0$ expresses a computation whose *result* is the maximal amount of information produced by that sequence. This is formalized as a limit. When the reduction is deterministic, it is standard to interpret such a limit as the meaning $[\![t]\!]$ of $t$. If however $t$ has *several possible reduction sequences*, each can produce a different outcome (a different limit). It is then natural to define the meaning $[\![t]\!]$ of a term $t$ as the *greatest* element in the set of limits, if any.[1] Intuitively, this means that the notion of "greatest amount of information produced by any reduction sequence" is well defined. To adopt such a notion demands care – for example, in the case of probabilistic and effectful computation, non-deterministic evaluation brings out issues which do not appear in pure lambda-calculus, not even when infinitary.

Given a term $t$ and a general reduction, the notion of result $[\![t]\!]$ is not necessarily defined: the set of limits for $t$ may contain different maximal elements, or it may not even have any maximal element (think of $\mathbb{N}$ or $[0, 1)$, which have no maximum). Maximal limits play a role similar to normal forms, and the following questions are then natural.

1. Is there a strategy that produces a maximal amount of information (a maximal limit)?
2. Given a term $t$, is $[\![t]\!]$ – the result of computing $t$ – well defined?

In Sect. 3 we provide tools to answer these questions, in this order, as we discuss next.

**On the workflow (and the limits of confluence).** The $\lambda$-calculus has two fundamental syntactical results: *confluence*, which implies **uniqueness of normal forms**, and the *standardization theorem,* which implies **normalization**, namely that a normal form (if any) can be reached by a computable strategy, which is a standard reduction (typically, left-to-right). Uniqueness guarantees that the notion of result is well defined, normalization provides a method to actually compute it.

A common workflow when studying $\lambda$-calculi is to first prove uniqueness of normal forms (via confluence), then normalization (via standardization). However, in an asymptotic setting *confluence does not directly imply* that the set of limits has a greatest element, but only that

[1] One could also define $[\![t]\!]$ as the lub of the set of limits, but this opens the question if there is a strategy that asymptotically computes $[\![t]\!]$, *internally* to the calculus. Since our focus is developing an *operational* theory, we require that $[\![t]\!]$ is itself a limit – it is a result that can be (asymptotically) computed.

it has a least upper bound. So, even if confluence is established, one still needs to prove that the lub is itself a limit, which may be a non-trivial task. For example, in the probabilistic $\lambda$-calculus [13, 14, 16], such a proof relies on (technical) properties of probability distributions.

In this paper, we *reverse the workflow*, and focus on normalization. In the finitary setting, if a rewriting relation $\to$ has a strategy $\underset{e}{\to} \subset \to$ that satisfies a suitable completeness hypothesis and uniqueness of normal forms, so does $\to$ (see [11]). With opportune definitions, this lifts well to the asymptotic setting. Forgoing confluence and focusing on normalization yields an efficient and uniform method which is easy to apply and which provides simultaneously (1.) existence and uniqueness of maximal limits, and (2.) a strategy to compute it.

**Content and contributions.** We start by illustrating asymptotic computation with examples (Sect. 1.1). Instances of asymptotic computation are quite diverse, and the syntax of each system may be rather complex. To study rewriting *abstractly*, in the spirit of Abstract Rewriting Systems (ARS), makes possible to analyze asymptotic properties in a way independent of a particular syntax, and to develop *general* proof techniques. In Sect. 2 we present the setting of Quantitative Abstract Rewriting Systems (QARS) [14], which are ARS enriched with a notion of observation. QARS are a natural refinement of ARSI [4].

Our first original contribution, and the heart of this paper, is Sect. 3, which proposes a proof technique to study asymptotic reduction strategies, and properties of the limits. We first introduce *asymptotic normalization*, which gives at the same time a tool to establish the existence of maximal limits – or of a greatest one – and a way to compute it. It formalizes the intuition that a normalizing strategy gradually computes (in a finite or infinite number of steps) the/a maximal amount of information that an element $t$ can produce. We then show (Sect. 3.1) that asymptotic normalization can be established by proving that a strategy is asymptotically complete and has a unique limit. Remarkably, such *infinitary properties* reduce to a finitary one, *factorization* (a simple form of *standardization*) and to some *local, elementary tests*, yielding a practical and versatile proof-technique.

We then apply our method to some representative case studies based on $\lambda$-calculus. In order to do so, we first revisit normalization for $\lambda$-calculus – uniformly for Call-by-Value and Call-by-Name – so as to have *a (novel) normalizing strategy* (Sect. 4.2) which is well-suited to asymptotic normalization, and to deal with (CbV and CbN) probabilistic $\lambda$-calculi. The application of our method to *probabilistic $\lambda$-calculus* yields a result of independent interest, which was left as open question in [16] (Remark 27 there), namely a theorem of asymptotic normalization for Call-by-Value probabilistic $\lambda$-calculus. We develop the CbV case explicitly in Sect. 5.1 – the same results hold *in a uniform way* for Call-by-Name. The same technique applies to other monadic calculi such as calculi with output (as we sketch in Sect. 6), but also to the asymptotic computation of Böhm trees, which can be obtained as the limit of a normalizing strategy (we leave this case to Appendix D.2).

## 1.1 Three examples of Asymptotic Computation

We illustrate three diverse examples of asymptotic computation, where the result of the computation is the limit of an *infinitary* process. All three examples are built on $\lambda$-calculus.

**Probabilistic computation.** A probabilistic program $P$ is a stochastic model generating a distribution over all possible outputs of $P$. Even if the termination probability is 1 (*almost sure termination*), that degree of certitude is typically not reached in a finite number of steps, but *as a limit*. A standard example is a term $M$ that reduces to either a normal form or

$M$ itself, with equal probability 1/2. After $n$ steps, $M$ is in normal form with probability $\frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^n}$. *Only at the limit* this computation terminates with probability 1. A direct way to model higher-order probabilistic computation is to endow the untyped $\lambda$-calculus with a binary operator $\oplus$ which models fair, binary probabilistic choice: $M_1 \oplus M_2$ reduces to either $M_1$ or $M_2$ with equal probability 1/2; we write this as $M_1 \oplus M_2 \to [\frac{1}{2}M_1, \frac{1}{2}M_2]$. Intuitively, *the result* of evaluating a probabilistic term is a *distribution* on its possible outputs.

▶ **Example 1.** Let $\Delta_\oplus = \lambda x.I \oplus (xx)$, where $I = \lambda x.x$. The term $M := \Delta_\oplus\Delta_\oplus$ has the behavior we have described above, and evaluates to $I$ with probability 1 only at the limit.

**Computations with output.** Consider a program that can print an output. Following [17], we can represent this with a pair $\mathtt{s} : M$, where $\mathtt{s}$ is a string over an alphabet $\mathbb{A}$, and $M$ is a term of the $\lambda$-calculus extended with a set of operators $\mathtt{out} = \{\mathtt{out}_c \mid c \in \mathbb{A}\}$. The term $\mathtt{out}_c(P)$ outputs $c$, adding it to the string, and continues as $P$. That is, $\langle \mathtt{s} : \mathtt{out}_c(P)\rangle \to \langle c.\mathtt{s} : P\rangle$.

▶ **Example 2.** Let $\mathbb{A} = \{0, 1\}$, and $\Delta_0 := \lambda x.\mathtt{out}_0(xx)$. The computation from $\langle \epsilon : \Delta_0\Delta_0\rangle$ (with $\epsilon$ the empty string) produces a stream: a string of 0's whose length tends to infinity.

**Infinite Normal Forms.** Infinitary $\lambda$-calculi [9, 22, 23] model infinite structures in $\lambda$-calculi. Terms and reduction sequences need not be finite. An infinite reduction sequence is *strongly convergent* if the depth of the contracted redex tends to infinity. Based on different depth measures, in [23] eight different infinitary $\lambda$-calculi are developed. If the calculus is *confluent*, the *infinite normal form of a term $N$ is unique*, and it is the *meaning of $N$*. Infinite normal forms are well-known in $\lambda$-calculus in the form of Böhm trees [7] or Lévy-Longo trees [28].

▶ **Example 3.** Let $\Delta_z := \lambda x.z(xx)$. In the (infinite) reduction sequence $\Delta_z\Delta_z \to_\beta z(\Delta_z\Delta_z) \to_\beta z(z(\Delta_z\Delta_z)) \to_\beta z(z(z(\Delta_z\Delta_z)))\ldots$, the depth of the redex $\Delta_z\Delta_z$ tends to infinity. It is intuitively clear that $\Delta_z\Delta_z$ has an *infinite normal form* $z(z(z\ldots))$.

**Notation.** From now on, we use the following standard notations: $I = \lambda x.x$, $\Delta = \lambda x.xx$, together with: $\Delta_\oplus = \lambda x.I \oplus (xx)$, $\Delta_c := \lambda x.\mathtt{out}_c(xx)$, $\Delta_z := \lambda x.z(xx)$.

## 1.2 Motivations, and necessity, for non-deterministic evaluation

In this paper we are concerned with evaluation towards a limit. We allow the evaluation $\underset{\mathsf{e}}{\to}$ (the normalizing strategy) to be *non-deterministic*. Let us discuss the motivations.

A programming language which is built on a $\lambda$-calculus implements a specific evaluation strategy $\underset{\mathsf{e}}{\to}$ of the general reduction $\to$. The evaluation strategy $\underset{\mathsf{e}}{\to}$ may or may not be deterministic, as long as *all choices eventually yield the same result.* Non-deterministic evaluation (written NDE) is a useful feature, which for example allows for parallel implementations, but in some cases is also a *necessity* and a key *reasoning tool*, as we discuss.

1. *NDE subsumes different evaluation policies.* A good illustration of this is in Plotkin's Call-by-Value $\lambda$-calculus, whose general reduction is $\to_{\beta_v}$. *Weak* evaluation (which does not reduce in the body of a function) evaluates closed terms to values. There are three main weak schemes (see Sect. 4.1): reducing left-to-right, as defined by Plotkin [30], right-to-left, as in Leroy's ZINC abstract machine [26], or in an arbitrary order. While left and right reduction are deterministic, weak reduction in arbitrary order is non-deterministic and *subsumes* both.
2. NDE *supports parallel/distributed implementation.* Non-deterministic evaluation does not define an abstract machine, but it includes *all possible parallel implementations.*

3. *NDE allows for breadth-first scheduling.* Left-to-right evaluation is inherently depth-first. NDE allows for breadth-first evaluation (favoring redexes at minimal depth), which is a necessity when the reduction graph is infinitary. An example comes from CbN $\lambda$-calculus. Thinking of Example 3, the terms $z(\Delta\Delta)(Iz)$ and $z(\Delta\Delta)(\Delta_z\Delta_z)$ do deliver more information than $z(\Delta\Delta)(\Delta\Delta)$. Their respective infinite normal forms are very different ($z\Omega z$ and $z\Omega(zz\dots)$, respectively). Still, for both left evaluation gets stuck at the leftmost redex, $\Delta\Delta$. Similar phenomena appear with effectful computation: in CbV $\lambda$-calculus with output of Example 2, the terms $(\Delta\Delta)(\mathtt{out}_0(I))$ and $(\Delta\Delta)(\Delta_0\Delta_0)$ behave similarly to the previous terms. A breadth-first approach allows one to compute the "best" (in some sense) possible result across all settings, uniformly.
4. *NDE facilitates reasoning and proofs.* This point is highly relevant when dealing with complex calculi, such as a probabilistic $\lambda$-calculus. Two examples from the literature are [16] and [10] – in both cases moving from the usual deterministic head reduction to its non-deterministic variant (given in Sect. 4.1) is crucial to the results.

## 2 Quantitative Abstract Rewriting Systems

In this section we present Quantitative Abstract Rewriting Systems (QARS) [4, 14]. QARS are Abstract Rewriting Systems (ARS) enriched with a notion of observation, where we can formalize both finitary and asymptotic rewriting. We first recall some standard notions of rewriting (see [31] or [6]), in particular that of ARS and of *normalizing strategy*.

### 2.1 Basics in (Finitary) Rewriting

An *abstract rewriting system (ARS)* is a pair $(\mathcal{A}, \to)$ consisting of a set $\mathcal{A}$ and a binary relation $\to$ on $\mathcal{A}$ whose pairs are written $t \to s$ and called *steps*. We denote $\to^*$ (resp. $\to^=$, $\to^+$) the transitive-reflexive (resp. reflexive, transitive) closure of $\to$. We write $t \leftarrow u$ if $u \to t$. If $\to_1, \to_2$ are binary relations on $\mathcal{A}$ then $\to_1 \cdot \to_2$ denotes their composition (*i.e.* $t \to_1 \cdot \to_2 s$ if there exists $u \in \mathcal{A}$ such that $t \to_1 u \to_2 s$). The relation $\to$ is *confluent* if $\leftarrow^* \cdot \to^* \subseteq \to^* \cdot \leftarrow^*$. An element $u \in \mathcal{A}$ is **$\to$-normal**, or a $\to$-*normal form* (*nf*) if there is no $t$ such that $u \to t$ (we also write $u \not\to$).

A **$\to$-sequence** (or **reduction sequence**) from $t$ is a possibly infinite sequence $t = t_0,\ t_1,\ t_2, \dots$ such that $t_i \to t_{i+1}$. Notice that $t \to^* s$ holds exactly when there is a *finite* sequence from $t$ to $s$ – we often write $t \to^* s$ to indicate a finite $\to$-sequence. A $\to$-sequence from $t$ is *maximal* if it is either infinite or ends in a $\to$-*nf*. We write $\langle t_n \rangle_n$ to indicate a maximal $\to$-sequence from $t_0$; by convention, if $t_i = u \not\to$ then $t_k = u$ for all $k \geq i$.

**Normalization.** In general, $t \in \mathcal{A}$ may or may not reduce to a normal form. And if it does, not all reduction sequences necessarily lead to normal form. $(\mathcal{A}, \to)$ is strongly (weakly, uniformly) normalizing if every $t \in \mathcal{A}$ is, where the normalization notions are as follows.

- $t$ is *strongly $\to$-normalizing*: every maximal $\to$-sequence from $t$ ends in a normal form;
- $t$ is *weakly $\to$-normalizing*: there is a $\to$-sequence from $t$ which ends in a normal form;
- $t$ is *uniformly $\to$-normalizing*: $t$ weakly $\to$-normalizing implies $t$ strongly $\to$-normalizing.

Untyped $\lambda$-calculus is not strongly normalizing. How do we compute a normal form, or *test* if any exists? This problem is tackled by *normalizing strategies*. By repeatedly performing *only specific steps* $\underset{\mathsf{e}}{\to}$, we are guaranteed that a normal form, if any, will eventually be computed.

A reduction $\underset{e}{\rightarrow}$ is a *one-step* (resp. *multi-step*) *strategy for* $\rightarrow$ if $\underset{e}{\rightarrow} \subseteq \rightarrow$ (resp. $\underset{e}{\rightarrow} \subseteq \rightarrow^{+}$), and it has the same normal forms as $\rightarrow$. It is a **normalizing strategy** for $\rightarrow$ if, moreover, whenever $t$ has a $\rightarrow$-normal form, then *every* maximal $\underset{e}{\rightarrow}$-sequence from $t$ ends in a $\rightarrow$-normal form. Note that $\rightarrow$ may not have the property of unique normal forms.

▶ Remark 4. A familiar example of calculus where terms may not have a unique normal form is Call-by-Name Weak $\lambda$-calculus (weak means no reduction under $\lambda$), studied by Abramsky and Ong [1]. The term $M = (\lambda xy.x)(II)$ has two distinct normal forms, $N_1 = \lambda y.II$ and $N_2 = \lambda y.I$. Weak head reduction is a normalizing strategy for it. However the strategy is not complete, in the sense that it produces the normal form $N_1$, but it cannot reach $N_2$.

A normalizing strategy $\underset{e}{\rightarrow}$ need not be deterministic (a reduction $\rightarrow$ is deterministic if for all $t \in \mathcal{A}$ there is at most one $s \in \mathcal{A}$ such that $t \rightarrow s$). However, $\underset{e}{\rightarrow}$ is required to be *uniformly normalizing*, *i.e.*, all reduction sequences from the same $t$ have the same behavior.

A property of $\underset{e}{\rightarrow}$ which guarantees uniform normalization is Newman's *Random Descent (RD)* [29]: for each $t \in \mathcal{A}$, all maximal sequences from $t$ have the same length and – if it is finite – they all end in the same element. The following property suffices to establish it.

▶ **Fact 5** (Newman). *If reduction $\underset{e}{\rightarrow}$ is RD-diamond, then it has Random Descent, where* RD-diamond*:* $(t_1 \underset{e}{\leftarrow} t \underset{e}{\rightarrow} t_2)$ *implies* $(t_1 = t_2$ *or* $\exists u.\ t_1 \underset{e}{\rightarrow} u \underset{e}{\leftarrow} t_2)$.

## 2.2 QARS

Ariola and Blom [4] have introduced the notion of Abstract Rewrite Systems with Information content (ARSI); a rewrite system is associated with a partial order that expresses the "information content" of the elements. ARSI however are tailored to infinite normal forms in the sense of Böhm and Levy-Longo trees: limits are there given by the ideal completion [3, Prop. 1.1.21] of the partial order. QARS [14] move from partial orders to *ω-complete partial orders* (ω-cpos) – this is enough to capture also effectful computation, such as the probabilistic one. We illustrate the key notions with several examples, including the calculi from Sect. 1.1.

Computation is a process that produces a result by gradually increasing the amount of available information – the standard structure to express a result in terms of partial information is that of an ω-cpo. Recall that a partially ordered set $\mathbb{S} = (\mathbb{S}, \leq)$ is an **ω-complete partial order (ω-cpo)** if every ω-chain $\boldsymbol{s}_0 \leq \boldsymbol{s}_1 \leq \dots$ has a supremum. We assume that $\leq$ has a *least element* $\bot$. The elements of $\mathbb{S}$ are denoted by bold letters $\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{q}$.

Let $(\mathcal{A}, \rightarrow)$ be an ARS. With each $t \in \mathcal{A}$ is associated a notion of (partial) information, called *observation*, by means of a function from $\mathcal{A}$ to an ω-cpo. Def. 6 formalizes this idea.

▶ **Definition 6** (QARS). *A* quantitative ARS *(*QARS*) is an ARS* $(\mathcal{A}, \rightarrow)$ *with a function* $\mathtt{obs}\colon \mathcal{A} \rightarrow \mathbb{S}$ *(where* $\mathbb{S}$ *is an ω-cpo) such that for all* $t, s \in \mathcal{A}$*, if* $t \rightarrow s$ *then* $\mathtt{obs}(t) \leq \mathtt{obs}(s)$*.*

Intuitively, the function obs observes a specific property of interest about $t \in \mathcal{A}$, and indicates how much stable information $t$ delivers: the information content is monotonically increasing during computation. Notice that obs may take numerical values, but needs not.

▶ **Example 7.**

1. $\lambda$-calculus: let $\mathbb{S} = \{0 < 1\}$ and $\mathtt{obs}_n(t) = 1$ if $t$ is normal, 0 otherwise.
2. Probabilistic $\lambda$-calculus: take $\mathbb{S} = ([0, 1], \leq_{\mathbb{R}})$, and for obs the probability to be in normal form (we will formalize this in Sect. 5 , see $\mathtt{obs}_{pn}(\mathtt{m})$ in Fig. 4.)
3. Infinitary $\lambda$-calculus: take $\mathbb{S} = \mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ with the usual order, and for obs the function which associates with any term $t$ the minimal depth $k$ of any redex in $t$.

▶ **Example 8** (Non-numerical obs).

1. $\lambda$-calculus: take for $\mathbb{S}$ the flat order on normal forms, and define $\mathtt{obs}_{\mathcal{N}}(u) = u$ if $u$ is normal, $\mathtt{obs}_{\mathcal{N}}(u) = \bot$ otherwise.
2. Probabilistic $\lambda$-calculus: take for $\mathbb{S}$ the $\omega$-cpo of the subdistributions on normal forms $\mathcal{D}(\mathcal{N})$ (we will formalize this in Sect. 5, see Fig. 4).
3. Infinitary $\lambda$-calculus: take the $\omega$-cpo of the partial normal forms that are associated with $\lambda$-terms (see [3] page 52, and Appendix D.2).

**Limits as Results.** From now on, let $\mathcal{Q} = ((\mathcal{A}, \to), \mathtt{obs})$ be an arbitrary but fixed QARS. By definition, given a $\to$-sequence $\langle t_n \rangle_n$, its *limit* $\sup_n\{\mathtt{obs}(t_n)\}$ with respect to obs always exists, because $\mathbb{S}$ is an $\omega$-cpo. If $\to$ is deterministic – hence any $t$ has a unique maximal $\to$-sequence – it is standard to interpret the limit as the *meaning* of $t$. In a QARS, $t$ has *several possible reduction sequences*, and so can produce several outcomes (limits). Following [13]:

▶ **Definition 9** (obs-limits). *Let $t \in \mathcal{A}$. We write*

- *$t \to^{\infty}_{\mathtt{obs}} \boldsymbol{p}$, if there exists a $\to$-sequence $\langle t_n \rangle_n$ from $t$ whose limit $\sup_n\{\mathtt{obs}(t_n)\} = \boldsymbol{p}$;*
- *$\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ is the set $\{\boldsymbol{p} \mid t \to^{\infty}_{\mathtt{obs}} \boldsymbol{p}\}$ of limits from $t$;*
- *$[\![t]\!]$ denotes the greatest element of $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$, if it exists.*

*The notations omit the subscript* obs *when the function* obs *is clear from the context.*

Intuitively, $[\![t]\!]$ is well defined if different reduction sequences from $t$ do not produce *essentially different* results: if $\boldsymbol{q} \neq \boldsymbol{p}$ then they both approximate a same result $\boldsymbol{r}$ (*i.e.*, $\boldsymbol{q}$, $\boldsymbol{p} \leq \boldsymbol{r}$).

Thinking of usual rewriting, consider $\mathtt{obs}_{\mathcal{N}}$ as in Example 8, point 1: here to have a greatest limit exactly corresponds to uniqueness of normal forms.

▶ **Example 10.** Let us revisit Example 7 pointwise, using the same notations.

1. $\lambda$-calculus: consider $t = (\lambda x.z)(\Delta\Delta)$. This term has infinite possible $\to_\beta$-sequences. The set of limits w.r.t. $\mathtt{obs}_n$ contains two elements: $\mathtt{Lim}_{\mathtt{obs}_n}(t, \to_\beta) = \{0, 1\}$
2. Probabilistic $\lambda$-calculus: consider the term $I \oplus \Delta\Delta$. It has only one reduction sequence $\mathtt{m} = [I \oplus \Delta\Delta] \Rightarrow [\frac{1}{2}I, \frac{1}{2}\Delta\Delta] \Rightarrow [\frac{1}{2}I, \frac{1}{2}\Delta\Delta] \Rightarrow \dots$. Here $\mathtt{Lim}_{\mathtt{obs}_{pn}}(\mathtt{m}, \Rightarrow) = \{\frac{1}{2}\}$.
3. Infinitary $\lambda$-calculus: consider the reduction sequence in Example 3. The depth of the redex $(\Delta_z\Delta_z)$ tends to $\infty$, which is the limit.

Note that maximal elements of $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ need not be maximal elements of $\mathbb{S}$. For instance, in Example 10.2, the term $I \oplus (\Delta\Delta)$ converges with probability $\frac{1}{2}$ (rather than 1). As a consequence, *the set of limits may or may not have maximal elements.* The fact that $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ may have a lub but not a maximum – similarly to $\mathbb{N}$ in $\mathbb{N}^\infty$ or the real interval $[0, 1)$ – is also easy to realize.

Even if $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ has maximal elements, a greatest limit does not necessarily exist: different reduction sequences may lead to different limits. The probabilistic $\lambda$-calculus and the $\lambda$-calculus with output provide several natural examples. Point 2 in Example 11 below shows moreover that the set of limits is – in general – *uncountable.*

▶ **Example 11** (Output $\lambda$-calculus). Consider the calculus sketched in Example 2. Let $\mathtt{Out}_{\mathbb{A}} = (\mathbb{A}^* \times \Lambda_{\mathtt{out}}, \underset{\mathsf{w}}{\to})$, where reduction is *CbV and weak*, with the obvious definitions. Let $\mathbb{S}$ be the $\omega$-cpo of strings, and let $\mathtt{obs}(\langle \mathtt{s} : M \rangle) = \mathtt{s}$. Clearly, $(\mathtt{Out}_{\mathbb{A}}, \mathtt{obs})$ is a QARS.

1. Let $\mathtt{m} = \langle \epsilon : \mathtt{out}_0(I)\mathtt{out}_1(I) \rangle$. $\mathtt{Lim}_{\mathtt{obs}}(\mathtt{m}, \underset{\mathsf{w}}{\to})$ contains two limits, 10 and 01, both maximal, because $\mathtt{m} \underset{\mathsf{w}}{\to} \langle 0 : I\mathtt{out}_1(I) \rangle \underset{\mathsf{w}}{\to} \langle 10 : II \rangle$, but also $\mathtt{m} \underset{\mathsf{w}}{\to} \langle 1 : \mathtt{out}_0(I)I \rangle \underset{\mathsf{w}}{\to} \langle 01 : II \rangle$.
2. Let $\mathtt{m}' = \langle \epsilon : M' \rangle$ for $M' = (\Delta_0\Delta_0)(\Delta_1\Delta_1)$. This produces all possible sequences on the alphabet $\{0, 1\}$. So $\mathtt{Lim}_{\mathtt{obs}}(\mathtt{m}', \underset{\mathsf{w}}{\to})$ has uncountable many elements, all maximal.

We are interested in the case when a greatest limit exists. The reason is that if $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ has a sup $\boldsymbol{s} \in \mathbb{S}$ which does not belong to $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$, no reduction sequence converges to $\boldsymbol{s}$; that is, we cannot compute $\boldsymbol{s}$ internally to the calculus.

## 3 Strategies and Asymptotic Normalization

The question of whether the result $[\![t]\!]$ of computing an element $t$ is well defined is natural. Equally natural is to wonder if there is a strategy that is guaranteed to compute $[\![t]\!]$. These two questions are at the core of this section. The existence of unique normal forms is independent of that of a normalizing strategy (see Remark 4). However, the computationally interesting case is (often) when both hold, so we will focus on this case.

We say that a reduction $\underset{\mathsf{e}}{\to} \subseteq \to$ is (asymptotically) normalizing if *each* $\underset{\mathsf{e}}{\to}$-sequence from a given $t$ converges *maximally*. We decompose this property in two properties: completeness and uniformity, which we discuss after the formal definition.

▶ **Definition 12** (Asymptotic properties). *Given a QARS* $\big((\mathcal{A}, \to), \mathtt{obs}\big)$*, a subreduction* $\underset{\mathsf{e}}{\to} \subseteq \to$ *is* ***asymptotically normalizing*** *for* $\to$ *(or* $\mathtt{obs}$*-normalizing) if it is both asymptotically complete and uniform, where*

1. $\underset{\mathsf{e}}{\to}$ *is* ***asymptotically complete*** *(or* $\mathtt{obs}$*-complete) if*
$$(\forall t \in \mathcal{A}) : t \to^{\infty}_{\mathtt{obs}} \boldsymbol{q} \text{ implies } t \underset{\mathsf{e}}{\to}^{\infty}_{\mathtt{obs}} \boldsymbol{p} \text{ for some } \boldsymbol{p} \text{ such that } \boldsymbol{q} \leq \boldsymbol{p};$$
2. $\underset{\mathsf{e}}{\to}$ *is* ***asymptotically uniform*** *(or* $\mathtt{obs}$*-uniform) if*
$$(\forall t \in \mathcal{A}) : \text{all elements in } \mathtt{Lim}_{\mathtt{obs}}(t, \underset{\mathsf{e}}{\to}) \text{ are maximal in } \mathtt{Lim}_{\mathtt{obs}}(t, \underset{\mathsf{e}}{\to}).$$

*All definitions adapt to* $\underset{\mathsf{e}}{\Rightarrow}$ *multistep subreduction of* $\to$*.*

Let us discuss all components, comparing with their ARS analog.

- *Completeness* guarantees that the strategy $\underset{\mathsf{e}}{\to}$ is as good as $\to$ in the amount of information it produces.
- *Completeness is not enough:* an asymptotically complete strategy is not guaranteed to find a/the "best" result: in Sect. 5.1 we will study a reduction $\underset{\mathrm{E}}{\Rightarrow}$ which is complete, but need not converge to the greatest limit (Remark 25). Let us first see a classical example.

▶ **Example 13.** In the usual $\lambda$-calculus (as in Example 10.1), the term $M = (\lambda x.I)(\Delta\Delta)$ has a $\to_\beta$-sequence which reaches $I$, and a diverging one. The leftmost-outermost strategy always produces $I$ (it is complete *and* normalizing). Notice that $\to_\beta$ is *trivially a complete strategy* for $\to_\beta$, but it is not normalizing, because $M$ has a diverging $\to_\beta$-sequence. Indeed, $\to_\beta$ is *complete, but not uniform.*

- *Asymptotic uniformity* expresses that all $\underset{\mathsf{e}}{\to}$-sequences from a term behave the same way. This corresponds to the ARS notion of *uniform normalization*: the reduction sequences from a term either all diverge, or all terminate (not necessarily in the same normal form).
- *Normalizing strategies.* If we consider usual ARS, and assume $\mathtt{obs}$ as in Example 7.1, expressing whether $t$ is or is not normal, then a strategy for $\to$ that is $\mathtt{obs}$-normalizing is exactly a normalizing strategy for $\to$ in the usual sense.

If $\underset{e}{\rightarrow} \subseteq \rightarrow$ is obs-*complete*, then $\mathtt{Lim}_{\mathtt{obs}}(t, \rightarrow)$ has maximal elements (resp. a greatest element) if and only if $\mathtt{Lim}_{\mathtt{obs}}(t, \underset{e}{\rightarrow})$ does. So we can reduce testing such properties for $\rightarrow$, to testing the same properties for $\underset{e}{\rightarrow}$, which is often simpler to study. In particular, if we are able to find a reduction $\underset{e}{\rightarrow} \subseteq \rightarrow$ which is complete and moreover has a unique limit, then necessarily $\rightarrow$ has a greatest limit. That is, we can simultaneously answer both of our questions: whether $[\![t]\!]$ is well defined, and if some strategy is guaranteed to compute it.

▶ **Proposition 14** (Main, abstractly). *If the following hold*

**i.** $\underset{e}{\rightarrow}$ *is asymptotically complete for* $\rightarrow$*;*

**ii.** $\mathtt{Lim}_{\mathtt{obs}}(t, \underset{e}{\rightarrow})$ *contains a unique element (*i.e. $\mathtt{Lim}_{\mathtt{obs}}(t, \underset{e}{\rightarrow}) = \{\boldsymbol{p}\}$*, for some* $\boldsymbol{p}$*).*

*Then: (1.)* $[\![t]\!]$ *is defined, and (2.)* $t \underset{e}{\rightarrow}^{\infty}_{\mathtt{obs}} [\![t]\!]$*, for each* $\underset{e}{\rightarrow}$*-sequence.*

Notice that condition (ii.) means that *all* $\underset{e}{\rightarrow}$-sequences from the term $t$ have the *same* limit.

▶ Remark 15 (Asymptotically normalizing strategies). If a QARS is such that $[\![t]\!]$ is defined for each $t$, then the two notions – to be an obs-normalizing strategy and to satisfy the conditions in Prop. 14 – coincide. Indeed, any obs-normalizing strategy for $\rightarrow$, if it exists, is forced to have a *unique* limit, that is, $\mathtt{Lim}_{\mathtt{obs}}(t, \underset{e}{\rightarrow}) = \{[\![t]\!]\}$.

## 3.1 A proof technique for Asymptotic Normalization

The two conditions in Prop. 14 give a method to prove normalization. The crucial step is to prove asymptotic completeness. Remarkably, as we show in this section, this can be reduced to prove a *finitary* property (*factorization*) and an elementary one-step test (*neutrality*).

The other condition in Prop. 14, namely uniqueness of limits, is trivial if the strategy is deterministic. Otherwise, random descent (opportunely formulated [14]) is a property that guarantees it, and that can also be established via a local test, as we recall below. While it is only a sufficient criterion, it often suffices to deal with non-deterministic evaluation strategies in $\lambda$-calculus, and in particular it suffices to deal with strategies in probabilistic $\lambda$-calculus.

**Asymptotic Completeness via Factorization.** The following theorem assumes a partition of the $\rightarrow$-steps into two classes: essential steps $\underset{e}{\rightarrow}$ and internal steps $\underset{\neg e}{\rightarrow}$. Point (i) states that every sequence $\rightarrow^*$ factorizes into a $\underset{e}{\rightarrow}$-sequence followed by a $\underset{\neg e}{\rightarrow}$-sequence. Point (ii) states that the internal steps $\underset{\neg e}{\rightarrow}$ do not increase the information content.

▶ **Theorem 16** (Asymptotic completeness criterion). *Given* $((A, \rightarrow), \mathtt{obs})$ *a QARS, and a subrelation* $\underset{e}{\rightarrow} \subseteq \rightarrow$*, assume :*

**i.** e-factorization*: if* $t \rightarrow^* u$ *then* $t \underset{e}{\rightarrow}^* \cdot \underset{\neg e}{\rightarrow}^* u$*;*

**ii.** $\neg$e-neutrality*:* $t \underset{\neg e}{\rightarrow} s$ *implies* $\mathtt{obs}(t) = \mathtt{obs}(s)$*.*

*Then:* $t \rightarrow^{\infty}_{\mathtt{obs}} \boldsymbol{p}$ *implies* $t \underset{e}{\rightarrow}^{\infty}_{\mathtt{obs}} \boldsymbol{p}$*.*

**Proof.** Let $\langle t_n \rangle_n$ be a $\rightarrow$-sequence such that $t = t_0$ and $\sup_n\{\mathtt{obs}(t_n)\} = \boldsymbol{p}$. From $t$, we inductively build a $\underset{e}{\rightarrow}$-sequence $\langle s_n \rangle_n$ with $s_0 = t$ and such that, for every $k \in \mathbb{N}$, there is an index $j(k)$ such that $t \underset{e}{\rightarrow}^* s_{j(k)}$ and $s_{j(k)} \underset{\neg e}{\rightarrow}^* t_k$. Case $k = 0$ is trivial (set $s_{j(0)} := t$).

Assume the claim holds for $k \geq 0$, so $t \underset{e}{\rightarrow}^* s_{j(k)}$. Observe that we have a sequence $s_{j(k)} \underset{\neg e}{\rightarrow}^* t_k \rightarrow t_{k+1}$. By applying assumption (i.) to it, we have $s_{j(k)} \underset{e}{\rightarrow}^* u \underset{\neg e}{\rightarrow}^* t_{k+1}$. We concatenate $t \underset{e}{\rightarrow}^* s_{j(k)}$ and $s_{j(k)} \underset{e}{\rightarrow}^* u$ to obtain $t \underset{e}{\rightarrow}^* s_{j(k)} \underset{e}{\rightarrow}^* s_{j(k+1)} := u$, as desired. By assumption (ii.), $s_{j(k)} \underset{\neg e}{\rightarrow}^* t_k$ implies $\mathtt{obs}(t_k) = \mathtt{obs}(s_{j(k)})$. The claim easily follows. ◀

**Uniqueness of the limit via Random Descent.** To establish that a strategy has a unique limit, Random Descent [29, 32, 33] has already been shown to adapt well and naturally in a probabilistic and asymptotic setting [13, 14].

The property obs-RD below states that if $t$ has different reduction sequences, they are all *indistinguishable* if regarded through the lenses of obs. Namely, all reduction sequences $\langle t_n \rangle_n$ starting from $t$ induce the same $\omega$-chain $\langle \mathtt{obs}(t_n) \rangle_n$. Thus, they all have the same obs-limit.

▶ **Definition 17** (Weighted Random Descent). *Let* $((\mathcal{A}, \to), \mathtt{obs})$ *be a QARS. The relation* $\underset{e}{\to} \subseteq \to$ *satisfies the following properties if they hold for each* $t \in \mathcal{A}$.

1. obs-*RD: for each pair of* $\underset{e}{\to}$*-sequences* $\langle r_n \rangle_n$, $\langle s_n \rangle_n$ *from* $t$, $\mathtt{obs}(r_n) = \mathtt{obs}(s_n)$ *for all* $n$.
2. obs-*diamond:* $\underset{e}{\to}$ *satisfies RD-diamond, and if* $t \underset{e}{\leftarrow} m \underset{e}{\to} s$ *then* $\mathtt{obs}(s) = \mathtt{obs}(t)$.

▶ **Proposition 18** ([14]). *With the same notation as in Def. 17:*
*(*obs*-diamond)* $\Rightarrow$ *(*obs*-RD)* $\Rightarrow$ $\mathtt{Lim}_{\mathtt{obs}}(t, \underset{e}{\to})$ *contains a unique element.*

▶ **Example 19** (CbV Weak reduction). Let us consider Call-by-Value $\lambda$-calculus with weak reduction $\underset{w}{\to}$, where weak means no reduction in the scope of $\lambda$-abstractions. The following are two different $\underset{w}{\to}$-sequences from the term $(II)(Ix)$:

$$(II)(Ix) \underset{w}{\to} I(Ix) \underset{w}{\to} Ix \underset{w}{\to} x \quad \text{and} \quad (II)(Ix) \underset{w}{\to} (II)x \underset{w}{\to} Ix \underset{w}{\to} x.$$

The observations of interest are values. Let $\mathtt{obs}_v : \Lambda \to \{0, 1\}$ be 1 if the term is a *value* (i.e. a variable or an abstraction), 0 otherwise. Through the lenses of $\mathtt{obs}_v$, both sequences appear as $\langle 0,\ 0,\ 0,\ 1 \rangle$.

## 4 Normalization in CbV and CbN $\lambda$-calculi

In the rest of the paper, we study asymptotic normalization in the setting of $\lambda$-calculi – in particular we are interested in probabilistic $\lambda$-calculus (Sect. 5).

In this section, after recalling the general syntax of $\lambda$-calculus, we define a novel, flexible normalizing strategy, which is uniformly defined for *Call-by-name* (CbN) and *Call-by-Value* (CbV) $\lambda$-calculi. Its features – in particular the fact that it support breadth-first reduction – make it suitable to then be extended to asymptotic normalization, in different settings.

### 4.1 Call-by-Name and Call-by-Value (applied) $\lambda$-calculus

We recall the basics of $\lambda$-calculus. Our syntax admits *operator symbols* [20, 30], *i.e. constants* with a fixed arity for their arguments. *Terms* and *values* are defined by the grammars below.

$$\begin{array}{lll} M & ::= & x \mid \lambda x.M \mid MM \mid \mathbf{o}(M, \ldots, M) \quad (\textit{Terms}, \Lambda_{\mathcal{O}}) \\ V & ::= & x \mid \lambda x.M \quad (\textit{Values}, \mathcal{V}) \end{array}$$

where $x$ ranges over a countable set of *variables*, and $\mathbf{o}$ over a disjoint (possibly empty) set $\mathcal{O}$ of operator symbols. If $\mathcal{O}$ is empty, the calculus is *pure* and we set $\Lambda := \Lambda_{\mathcal{O}}$. Terms are identified up to renaming of bound variables, where $\lambda x$ is the only binder constructor. $P\{Q/x\}$ is the capture-avoiding substitution of $Q$ for the free occurrences of $x$ in $P$.

**Contexts** (with an hole $(\!|\ |\!)$) are defined by the grammar below. $\mathbf{C}(\!|N|\!)$ stands for the term obtained from $\mathbf{C}$ by replacing the hole with $N$ (possibly capturing the free variables of $N$).

$$\mathbf{C} ::= (\!|\ |\!) \mid M\mathbf{C} \mid \mathbf{C}M \mid \lambda x.\mathbf{C} \mid \mathbf{o}(M, \ldots, \mathbf{C}, \ldots, M) \qquad (\textit{Contexts})$$

**Rules and Reductions.** A **rule** $\rho$ is a binary relation on $\Lambda_{\mathcal{O}}$, which we also denote $\mapsto_\rho$, writing $R \mapsto_\rho R'$. $R$ is called a *$\rho$-redex*. The best known rule is $\beta$: $(\lambda x.M)N \mapsto_\beta M\{N/x\}$.

A **reduction step** $\to_\rho$ is the closure under context $\mathsf{C}$ of $\rho$.

**CbN and CbV Calculi.** The (pure) **Call-by-Name** calculus $\Lambda^{\mathsf{cbn}} = (\Lambda, \to_\beta)$ is the set of terms equipped with the contextual closure of the $\beta$-rule, as described *e.g.* in [7]. The (pure) **Call-by-Value** calculus $\Lambda^{\mathsf{cbv}} = (\Lambda, \to_{\beta_v})$ is the same set equipped with the contextual closure of the $\beta_v$-rule: $(\lambda x.M)V \mapsto_{\beta_v} M\{V/x\}$ where $V \in \mathcal{V}$, as introduced by Plotkin [30].

**CbN and CbV applied** calculi are obtained by associating to operators (the contextual closure of) a family of rules of the form $\mathbf{o}(M_1, \dots, M_k) \mapsto_{\mathbf{o}} N$. This is a standard way to enrich $\lambda$-calculus with new computational features, such as probabilistic choice or output.

**Weak reductions in CbV.** In CbV $\lambda$-calculus, various restrictions of $\to_{\beta_v}$ are studied. If the result of interest are values, the reduction is *weak*, that is, it does not reduce in the body of a function. There are three main weak schemes: left, right and in arbitrary order. *Left* contexts $\mathsf{L}$, *right* contexts $\mathsf{R}$, and (arbitrary order) *weak* contexts $\mathsf{W}$ are defined by

$$\mathsf{L} ::= (\!|\ |\!) \mid \mathsf{L}M \mid V\mathsf{L} \qquad \mathsf{R} ::= (\!|\ |\!) \mid M\mathsf{R} \mid \mathsf{R}V \qquad \mathsf{W} ::= (\!|\ |\!) \mid \mathsf{W}M \mid M\mathsf{W}$$

Given a rule $\mapsto$ on $\Lambda$, *weak reduction* $\underset{\mathsf{w}}{\to}$ is the closure of $\mapsto$ under context $\mathsf{W}$. A step $T \to S$ is *non-weak*, noted $T \underset{\neg\mathsf{w}}{\to} S$ if it is not weak. Similarly for left ($\underset{\mathsf{l}}{\to}$ and $\underset{\neg\mathsf{l}}{\to}$), and right ($\underset{\mathsf{r}}{\to}$ and $\underset{\neg\mathsf{r}}{\to}$). Left and right reduction are *deterministic*. Reduction $\underset{\mathsf{w}}{\to}_{\beta_v}$ *subsumes* both. The choice of a redex is non-deterministic, but irrelevant w.r.t. reaching a value and the number of steps to do so, because $\underset{\mathsf{w}}{\to}_{\beta_v}$ is RD-diamond (Fact 5). We can fire any arbitrary redex in weak position – or *all of them* in parallel. A *parallel variant* can easily be defined.

Weak factorization holds for the three reductions: $\to_{\beta_v}{}^* \subseteq \underset{\mathsf{s}}{\to}_{\beta_v}{}^* \cdot \underset{\neg\mathsf{s}}{\to}_{\beta_v}{}^*$, for $\mathsf{s} \in \{\mathsf{w}, \mathsf{l}, \mathsf{r}\}$.

**Head reduction in CbN.** Head reduction [7] is the closure of $\beta$ under head context $\lambda x_1 \dots x_n.(\!|\ |\!)M_1 \dots M_k$. *Head normal forms* (*hnf*), whose set is denoted by $\mathcal{H}$, are its normal forms. The literature of linear logic often uses a variant of head context which includes the standard one, and induces exactly the same set $\mathcal{H}$ of normal forms. Given a rule $\rho$, we write $\underset{\mathsf{h}}{\to}_\rho$ for its closure under context $\mathsf{H}$.

$$\mathsf{H} ::= (\!|\ |\!) \mid \lambda x.\mathsf{H} \mid \mathsf{H}M \qquad (\textit{Head contexts})$$

Head factorization (see [7, Lemma 11.4.6]) and head normalization (see [7, Thm. 8.3.11]) are classical results, which hold also when the calculus includes constants, *i.e.* for $(\Lambda_{\mathcal{O}}, \to_\beta)$.

## 4.2 A strategy for finitary normalization in CbV and CbN $\lambda$-calculus

We revisit normalization for $\lambda$-calculus – uniformly for CbV and CbN – and define a strategy which is well-suited to be extended to probabilistic $\lambda$-calculi, and to asymptotic normalization. It supports non-deterministic head and weak reduction (as needed in the probabilistic case) and breadth-first evaluation of redexes (as needed to deal with infinitary reduction graphs).

We call *surface reduction* weak reduction in CbV and head reduction in CbN, because they only fire redexes at *depth 0*, where in CbV the depth of a redex $R$ is the number of *abstractions* in which $R$ is nested, and in CbN is the number of arguments. Normal forms for $\beta$ and $\beta_v$ can be computed by iterating surface reduction in a suitable way, as we show below.

**Normalizing strategies.** In $\Lambda^{\mathsf{cbn}}$, a paradigmatic normalizing strategy is leftmost-outermost reduction. It can be described as: first apply head reduction $\underset{\mathsf{h}}{\rightarrow}_\beta$ until *hnf*, and then iterate the process, in left-to-right order. Normalization in $\Lambda^{\mathsf{cbv}}$ is less established: one can iterate $\underset{\mathsf{l}}{\rightarrow}_{\beta_v}$ left to right (as in Plotkin's standard reduction [30]), but also iterate $\underset{\mathsf{r}}{\rightarrow}_{\beta_v}$ right to left, as in Grégoire and Leroy's implementation [19]. In all cases, once a head or weak normal form is reached (think of $xM_1 \dots M_k$ in CbN) no interaction is possible among the subterms $M_i, \dots, M_k$, so in fact the process can be iterated in any *arbitrary order*.

We define a rather liberal normalizing strategy, *uniformly* for CbN and CbV, and *parametrically* in the choice of surface reduction. Unlike leftmost-outermost reduction, which is sequential and *inherently depth-first*, the *unbiased* reduction $\underset{U}{\rightarrow}$ is non-deterministic in the choice of the outermost redex, and can support a breadth-first reduction policy. It persistently performs surface steps, as long as it is possible, and then iterates the process in the subterms, in arbitrary order.

▶ **Definition 20** (Unbiased iteration of surface reduction)**.** *Given* $(\Lambda_{\mathcal{O}}, \rightarrow)$*, where* $\rightarrow$ *is the contextual closure of a rule* $\mathsf{b} \in \{\beta, \beta_v\}$*, let* $\underset{\mathsf{s}}{\rightarrow} \subseteq \rightarrow$ *be as follows:*

$$\underset{\mathsf{s}}{\rightarrow} = \underset{\mathsf{h}}{\rightarrow} \ \textit{if } \mathsf{b} = \beta \ (\textit{CbN}) \qquad \underset{\mathsf{s}}{\rightarrow} \in \{\underset{\mathsf{w}}{\rightarrow}, \underset{\mathsf{l}}{\rightarrow}, \underset{\mathsf{r}}{\rightarrow}\} \ \textit{if } \mathsf{b} = \beta_v \ (\textit{CbV}) \ .$$

*The relation* $\underset{U}{\rightarrow} \subseteq \rightarrow$ *is inductively defined as follows:*

- *if* $M \underset{\mathsf{s}}{\rightarrow} M'$ *then* $M \underset{U}{\rightarrow} M'$*;*
- *if* $M \underset{\mathsf{s}}{\not\rightarrow}$ *then* $M \underset{U}{\rightarrow} M'$ *is defined according the rules below.*

$$\frac{P \underset{U}{\rightarrow} P'}{(\lambda x.P) \underset{U}{\rightarrow} (\lambda x.P')} \quad \frac{P \underset{U}{\rightarrow} P'}{PQ \underset{U}{\rightarrow} P'Q} \quad \frac{Q \underset{U}{\rightarrow} Q'}{PQ \underset{U}{\rightarrow} PQ'} \quad \frac{P_i \underset{U}{\rightarrow} P_i'}{\mathbf{o}(P_1, \dots, P_i, \dots, P_k) \underset{U}{\rightarrow} \mathbf{o}(P_1, \dots, P_i', \dots, P_k)}$$

*The same definition of* $\underset{U}{\rightarrow} \subseteq \rightarrow$ *still applies if* $\rightarrow$ *is the contextual closure of* $\mapsto_{\mathsf{b}} \cup \mapsto_\rho$*, i.e. of the rule* $\mapsto_{\mathsf{b}}$ *extended with some other rule* $\mapsto_\rho$ *on* $\Lambda_{\mathcal{O}}$*.*

We study $\underset{U}{\rightarrow}_{\mathsf{b}}$. It is RD-diamond (see Fact 5) and is a normalizing strategy for both CbN and CbV $\lambda$-calculi. Note that in CbN, $\underset{U}{\rightarrow}_\beta$ *subsumes* usual *leftmost-outermost reduction.*

▶ **Proposition 21** (U-Factorization)**.** *Let* $\mathsf{b} \in \{\beta, \beta_v\}$*.*

$$M \rightarrow_{\mathsf{b}}{}^* N \textit{ implies } M \underset{U}{\rightarrow}_{\mathsf{b}}{}^* \cdot \underset{\neg U}{\rightarrow}_{\mathsf{b}}{}^* N \qquad \text{(U-Factorization)}$$

▶ **Proposition 22.** *With the same assumptions as in Def. 20, let* $\mathsf{b} \in \{\beta, \beta_v\}$*. Then:*

1. $\underset{U}{\rightarrow}_{\mathsf{b}}$ *is RD-diamond.*
2. $\underset{U}{\rightarrow}_{\mathsf{b}}$ *has the same normal forms as* $\rightarrow_{\mathsf{b}}$*.*
3. *Let* $N$ *be* $\mathsf{b}$*-normal.* $M \rightarrow_{\mathsf{b}}{}^* N$ *implies* $M \underset{U}{\rightarrow}_{\mathsf{b}}^* N$*.*

Normalization for both CbN and CbV follows from the points above.

▶ **Theorem 23** (Normalization)**.** *For* $\mathsf{b} \in \{\beta, \beta_v\}$*,* $\underset{U}{\rightarrow}_{\mathsf{b}}$ *is a normalizing strategy for* $\rightarrow_{\mathsf{b}}$*.*

**Depth-first vs Breadth-first.** Leftmost-outermost reduction fires redexes in a depth-first way. Instead, $\underset{U}{\rightarrow}$ evaluates in a breadth-first style, which is more suitable to deal with possibly infinitary reductions. For example, in CbN think of $z(\Delta\Delta)(\Delta_z\Delta_z)$. Leftmost-outermost reduction never leaves the redex $\Delta\Delta$, while $\underset{U}{\rightarrow}$ can also fire $(\Delta_z\Delta_z)$ yielding $z(z(z \dots))$.

**A parallel variant.** Once a term is $\underset{s}{\rightarrow}$-normal, the process can be iterated in any arbitrary order, or in *parallel*. Parallel (multi-step) reduction $\underset{/\!/\mathsf{U}}{\rightarrow}$ is easily defined (Appendix B.1).

## 5 Probabilistic $\lambda$-calculi and Asymptotic Normalization

A standard way to model *probabilistic choice* (a fair coin) is by means of a binary operator $\oplus$. We write $M \oplus N$ for $\oplus(M, N)$. Intuitively, $M \oplus N$ reduces to either $M$ or $N$, *with equal probability* $\frac{1}{2}$. Reduction is then defined not simply on terms but on (monadic) structures representing probability distributions over terms. Here we follow [16], which defines both a CbV and a CbN calculus $\Lambda_\oplus^{\mathsf{cbv}}$ and $\Lambda_\oplus^{\mathsf{cbn}}$, where $\beta$ or $\beta_v$ reduction are "as usual", so if a term contains no probabilistic operator, it behaves the same as in the usual $\lambda$-calculus (*i.e.* the extension is *conservative*). Probabilistic reduction instead needs to be constrained in order to have good properties such as confluence (see [16], and [12,25] for a discussion of the issues).

**Discrete Probability Distributions.** Given a countable set $\Omega$, a function $\mu \colon \Omega \to [0,1]$ is a probability *subdistribution* if $\|\mu\| := \sum_{\omega\in\Omega} \mu(\omega) \le 1$ (a *distribution* if $\|\mu\| = 1$). Subdistributions allow us to deal with partial results. We write $\mathcal{D}(\Omega)$ for the set of subdistributions on $\Omega$, equipped with the pointwise order on functions: $\mu \le \rho$ if $\mu(\omega) \le \rho(\omega)$ for all $\omega \in \Omega$. $\mathcal{D}(\Omega)$ has a bottom element (the subdistribution $\mathbf{0}$) and maximal elements (all distributions).

**Multi-distributions.** We use multi-distributions [5] to syntactically represent distributions, A *multi-distribution* $\mathtt{m} = [p_i M_i]_{i\in I}$ on the set of terms $\Lambda_\mathcal{O}$ is a finite multiset of pairs of the form $pM$, with $p \in ]0,1]$, $M \in \Lambda_\mathcal{O}$, and $\sum_i p_i \le 1$. The set of all multi-distributions on $\Lambda_\mathcal{O}$ is $\mathcal{M}(\Lambda_\mathcal{O})$. The sum of multi-distributions is noted $+$. The product $q \cdot \mathtt{m}$ of a scalar $q$ and a multi-distribution $\mathtt{m}$ is defined pointwise $q[p_i M_i]_{i\in I} := [(qp_i)M_i]_{i\in I}$. We write $[M]$ for $[1M]$.

**Syntax.** *Terms* ($\Lambda_\oplus$) and *values* are as in Sect. 4.1, with the operator $\mathbf{o}$ being here $\oplus$.

**Call-by-Value.** The *calculus* $\Lambda_\oplus^{\mathsf{cbv}}$ is the rewrite system $(\mathcal{M}(\Lambda_\oplus), \Rightarrow)$ where $\mathcal{M}(\Lambda_\oplus)$ is the set of *multi-distributions* on $\Lambda_\oplus$ and the relation $\Rightarrow \subseteq \mathcal{M}(\Lambda_\oplus) \times \mathcal{M}(\Lambda_\oplus)$ is defined in Fig. 1 and Fig. 2. First, define one-step reductions from terms to multi-distributions – so for example, $M \oplus N \to [\frac{1}{2}M, \frac{1}{2}N]$. Then, lift the definition of reduction to a binary relation on $\mathcal{M}(\Lambda_\oplus)$, in the natural way – for instance $[\frac{1}{2}(\lambda x.x)z, \frac{1}{2}(M \oplus N)] \Rightarrow [\frac{1}{2}z, \frac{1}{4}M, \frac{1}{4}N]$. Precisely:

1. The reductions $\to_{\beta_v}, \to_\oplus \subseteq \Lambda_\oplus \times \mathcal{M}(\Lambda_\oplus)$ are defined in Fig. 1. *Contexts* **C** *and* **W** are as in Sect. 4.1. Note that $\beta_v$ is closed under arbitrary context, while the $\oplus$ rule – probabilistic choice – is closed under weak contexts **W** (no reduction in the scope of $\lambda$ or $\oplus$). We write $\underset{s}{\rightarrow}_{\beta_v}$ for the closure of $\beta_v$ under context **W**. The relation $\to$ is $\to_{\beta_v} \cup \to_\oplus$. *Surface reduction* is $\underset{s}{\rightarrow} \; = \; \underset{s}{\rightarrow}_{\beta_v} \cup \to_\oplus$. A $\to$-step which is not surface is noted $\underset{\neg s}{\rightarrow}$.
2. The lifting of a relation $\to_r \subseteq \Lambda_\oplus \times \mathcal{M}(\Lambda_\oplus)$ to a reduction on multi-distributions is defined in Fig. 2. In particular, $\to, \to_{\beta_v}, \to_\oplus, \underset{s}{\rightarrow}, \underset{\neg s}{\rightarrow}$ lift to $\Rightarrow, \Rightarrow_{\beta_v}, \Rightarrow_\oplus, \underset{s}{\Rightarrow}, \underset{\neg s}{\Rightarrow}$.

A term $M$ is $\to$**-normal** if there is no $\mathtt{m}$ such that $M \to \mathtt{m}$. We also write $M \not\to$. We denote by $\mathcal{N}_v$ the set of the **normal forms** of $\to = (\to_{\beta_v} \cup \to_\oplus)$.

**Call-by-Name.** The calculus $\Lambda_\oplus^{\mathsf{cbn}}$ is defined in a similar way, by replacing $\beta_v$ with $\beta$ and weak contexts with head contexts **H** (as defined in Sect. 4.1).

$$\mathbf{C}(\!|(\lambda x.M)V|\!) \to_{\beta_v} [\mathbf{C}(\!|M\{V/x\}|\!)]$$
$$\mathbf{W}(\!|M \oplus N|\!) \to_{\oplus} [\frac{1}{2}\mathbf{W}(\!|M|\!), \frac{1}{2}\mathbf{W}(\!|N|\!)]$$
$$\to := \to_{\beta_v} \cup \to_{\oplus} \qquad \underset{s}{\to} := \underset{s}{\to}_{\beta_v} \cup \to_{\oplus}$$

**Figure 1** $\to$-steps for the calculus $\Lambda_{\oplus}^{\mathsf{cbv}}$.

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

**Figure 2** Lifting of $\to$.

$$\frac{M \not\to}{[M] \rightrightarrows [M]} \qquad \frac{M \to \mathtt{m}}{[M] \rightrightarrows \mathtt{m}} \qquad \frac{([M_i] \rightrightarrows \mathtt{m}_i)_{i\in I}}{[p_i M_i \mid i \in I] \rightrightarrows \sum_{i\in I} p_i \cdot \mathtt{m}_i}$$

**Figure 3** Full lifting of $\to$.

$$\mathtt{obs}_{\mathcal{N}} : \mathcal{M}(\Lambda_\oplus) \to \mathcal{D}(\mathcal{N}_v) \quad [p_i M_i]_{i\in I} \mapsto \mu \quad \text{where } \forall N \in \mathcal{N}_v,\ \mu(N) = \textstyle\sum_{i\in I} p_i \text{ s.t. } M_i = N$$
$$\mathtt{obs}_{pn} : \mathcal{M}(\Lambda_\oplus) \to [0,1] \qquad [p_i M_i]_{i\in I} \mapsto \|\mu\|$$

**Figure 4** CbV observations on multi-distributions.

**Observations on multi-distributions.** In **CbV**, events of interest are the set $\mathcal{V}$ of values and the set $\mathcal{N}_v$ of $\to$-normal forms (for $\to = \to_{\beta_v} \cup \to_\oplus$). Focusing on $\mathcal{N}_v$, we can define:

- $\mathtt{obs}_{\mathcal{N}}$ extracts from $\mathtt{m} = [p_i M_i]_{i\in I}$ a *subdistribution* $\mu$ *over normal forms.* For example, if $\mathtt{m} = [\frac{1}{4}\mathtt{T}, \frac{1}{8}\mathtt{T}, \frac{1}{4}\mathtt{F}, \frac{1}{4}II]$, $\mathtt{obs}_{\mathcal{N}}(\mathtt{m})$ is the subdistribution $\{\mathtt{T}^{\frac{3}{8}}, \mathtt{F}^{\frac{1}{4}}\}$, *i.e.* $\mu(\mathtt{T}) = \frac{3}{8}$, $\mu(\mathtt{F}) = \frac{1}{4}$.
- $\mathtt{obs}_{pn}$ observes the probability that $\mathtt{m}$ has reached a normal form. For example, with $\mathtt{m}$ as above, $\mathtt{obs}_{pn}(\mathtt{m}) = \frac{5}{8}$.

In **CbN**, events of interest are the set of normal forms (w.r.t. $\to_\beta \cup \to_\oplus$), and the set $\mathcal{H}$ of head normal forms. The corresponding observations are defined in the obvious way.

## 5.1 Asymptotic Normalization for Probabilistic $\lambda$-Calculi

We can now revisit the probabilistic calculi $\Lambda_{\oplus}^{\mathsf{cbv}}$ and $\Lambda_{\oplus}^{\mathsf{cbn}}$ as QARS, and define for them an asymptotically normalizing strategy. We develop explicitly only the CbV case, but similar definitions and results hold for CbN, taking into account that $\to_{\beta_v}$ is replaced by $\to_\beta$ and surface reduction is $\underset{h}{\to}$. Method and proofs are exactly the same.

The QARS framework allows us to express and analyze the asymptotic behaviour of the calculus $\Lambda_{\oplus}^{\mathsf{cbv}} = (\mathcal{M}(\Lambda_\oplus), \Rightarrow)$. Here we are interested in $\mathtt{obs}_{\mathcal{N}} : \mathcal{M}(\Lambda_\oplus) \to \mathcal{D}(\mathcal{N})$ as defined in Fig. 4. It is immediate that $\mathtt{m} \to \mathtt{m}'$ implies $\mathtt{obs}_{\mathcal{N}}(\mathtt{m}) \leq \mathtt{obs}_{\mathcal{N}}(\mathtt{m}')$. So, $\left(\Lambda_{\oplus}^{\mathsf{cbv}}, \mathtt{obs}_{\mathcal{N}}\right)$ is a QARS. We prove (Thm. 31) that $\Lambda_{\oplus}^{\mathsf{cbv}}$ satisfies the following properties: (1) the result $[\![\mathtt{m}]\!]$ of computing $\mathtt{m}$ is well defined; (2) there exists a strategy that is guaranteed to produce $[\![\mathtt{m}]\!]$.

**Beyond the surface.** We define a reduction $\underset{E}{\to} \subseteq \Lambda_\oplus \times \mathcal{M}(\Lambda_\oplus)$ which performs surface steps ($\underset{s}{\to} = \underset{s}{\to}_{\beta_v} \cup \to_\oplus$, see Sect. 5) as much as possible, and then iterates the process on the subterms. There are two subtleties here. First: $M \underset{s}{\not\to}$ if and only if ($M \underset{s}{\not\to}_{\beta_v}$ and $M \underset{s}{\not\to}_\oplus$). Second: an occurrence of $\oplus$-redex can only be fired when it is a surface redex. By keeping this into account, Def. 20 updates as follows. We denote by $\mathcal{S}$ the set of $\underset{s}{\to}$**-normal forms**.

▶ **Definition 24** (Unbiased evaluation $\underset{E}{\Rightarrow}$, $\underset{E}{\rightrightarrows}$ )**.**

- *The relation* $\underset{E}{\to} \subseteq \Lambda_\oplus \times \mathcal{M}(\Lambda_\oplus)$ *is defined by the following rules, depending if* $M \notin \mathcal{S}$ *or* $M \in \mathcal{S}$*. The relation* $\underset{U}{\to}_{\beta_v}$ *is as in Def. 20.*

$$\frac{M \underset{s}{\to} \mathtt{m}}{M \underset{E}{\to} \mathtt{m}}\ (M \notin \mathcal{S}) \qquad \frac{M \underset{s}{\not\to} \quad M \underset{U}{\to}_{\beta_v} M'}{M \underset{E}{\to} [M']}\ (M \in \mathcal{S})$$

- $\underset{E}{\Rightarrow}, \underset{E}{\rightrightarrows} \subseteq \mathcal{M}(\Lambda_\oplus) \times \mathcal{M}(\Lambda_\oplus)$ *are respectively the lifting and full lifting of* $\underset{E}{\to}$ *(Figs. 2 and 3).*

Clearly, $\underset{\mathrm{E}}{\rightarrow} \subseteq \rightarrow$, and moreover $\rightarrow$ and $\underset{\mathrm{E}}{\rightarrow}$ have the *same normal forms.*

▶ Remark 25. $\underset{\mathrm{E}}{\Rrightarrow}$ is $\mathtt{obs}_{\mathcal{N}}$-complete, but not $\mathtt{obs}_{\mathcal{N}}$-normalizing for $\Rightarrow$. Indeed, the sequence $\mathtt{m} = [II \oplus \Delta\Delta] \underset{\mathrm{E}}{\Rrightarrow} [\frac{1}{2}II, \frac{1}{2}\Delta\Delta] \underset{\mathrm{E}}{\Rrightarrow} [\frac{1}{2}II, \frac{1}{2}\Delta\Delta] \underset{\mathrm{E}}{\Rrightarrow} \dots$ never fires $II$. The solution is to move to $\underset{\mathrm{E}}{\rightrightarrows}$, which forces all non-normal terms to reduce. Note that $\underset{\mathrm{E}}{\rightrightarrows}$ does not factorize $\Rightarrow$.

We show that $\underset{\mathrm{E}}{\rightrightarrows}$ is an $\mathtt{obs}_{\mathcal{N}}$-normalizing strategy for $\Rightarrow$. The pillars of our construction are E-factorization and weighted Random Descent. The former holds for $\underset{\mathrm{E}}{\Rrightarrow}$, the latter for $\underset{\mathrm{E}}{\rightrightarrows}$.

▶ **Proposition 26** (Factorization and $\mathtt{obs}_{\mathcal{N}}$-neutrality)**.**
1. E-factorization: $\mathtt{m} \Rightarrow^* \mathtt{n}$ *implies* $\mathtt{m} \underset{E}{\Rrightarrow}^* \cdot \underset{\neg E}{\Rrightarrow}^* \mathtt{n}$.
2. $\mathtt{obs}_{\mathcal{N}}$-neutrality: $\mathtt{m} \underset{\neg E}{\Rrightarrow} \mathtt{n}$ *implies* $\mathtt{obs}_{\mathcal{N}}(\mathtt{m}) = \mathtt{obs}_{\mathcal{N}}(\mathtt{n})$.

▶ **Proposition 27** (Diamond)**.** $\underset{E}{\rightrightarrows}$ *is* $\mathtt{obs}_{\mathcal{N}}$*-diamond.*

We are now ready to prove that, in $(\Lambda_{\oplus}^{\mathtt{cbv}}, \mathtt{obs}_{\mathcal{N}})$, the reduction $\underset{\mathrm{E}}{\rightrightarrows} \subseteq \Rightarrow$ (*i.e.*, the full lifting of $\underset{\mathrm{E}}{\rightarrow}$) is guaranteed to compute the best possible result from each $\mathtt{m} \in \mathcal{M}(\Lambda_{\oplus})$.

**Asymptotic Completeness.** We have that $\underset{\mathrm{E}}{\Rrightarrow}$ is asymptotically complete for $\Rightarrow$, because it satisfies the conditions of Thm. 16 (by Prop. 26).

▶ **Lemma 28.** *If* $\mathtt{m} \Rightarrow^{\infty} \mathtt{r}$ *then* $\mathtt{m} \underset{E}{\Rrightarrow}^{\infty} \mathtt{r}$.

In turn, $\underset{\mathrm{E}}{\rightrightarrows}$ is asymptotically complete for $\underset{\mathrm{E}}{\Rrightarrow}$ (immediate). So via Lemma 28 we have:

▶ **Theorem 29.** $\underset{E}{\rightrightarrows}$ *is asymptotically complete for* $\Rightarrow$*: if* $\mathtt{m} \Rightarrow^{\infty} \mathtt{r}$ *then* $\mathtt{m} \underset{E}{\rightrightarrows}^{\infty} \mathtt{s}$ *and* $\mathtt{r} \leq \mathtt{s}$.

**Unique Result.** All $\underset{\mathrm{E}}{\rightrightarrows}$-sequences from $\mathtt{m}$ converge to the same limit, by Prop. 18 and 27.

▶ **Theorem 30.** $\mathtt{Lim}_{\mathtt{obs}_{\mathcal{N}}}(\mathtt{m}, \underset{E}{\rightrightarrows})$ *contains a unique element.*

**Asymptotic Normalization.** By Prop. 14, the main result follows from Thm. 29 and 30.

▶ **Theorem 31** (Main, probabilistic CbV)**.** *For each* $\mathtt{m} \in \mathcal{M}(\Lambda_{\oplus})$*:*
1. $[\![\mathtt{m}]\!]$ *is defined;*
2. $\mathtt{m} \underset{E}{\rightrightarrows}^{\infty} \mathtt{r}$ *if and only if* $\mathtt{r} = [\![\mathtt{m}]\!]$.

*Hence* $\underset{E}{\rightrightarrows}$ *is an* $\mathtt{obs}_{\mathcal{N}}$-normalizing strategy *for* $\Rightarrow$ *(see Remark 15).*

Some simple examples will help to see how the normalizing strategy works, and how it differs from surface reduction.

▶ **Example 32.** Recall that $\beta_v$-reduction is unrestricted, so for example $M = \lambda z.(Iz) \to_{\beta_v} \lambda z.z$. Instead, $\lambda z.(Iz) \not\to_{\mathsf{s}}$, because surface reduction cannot fire under abstraction. So *surface reduction is not a complete strategy* w.r.t. $\beta_v$-normal forms.

A direct consequence is that surface reduction is not informative about normalization, as it produces "false positive". For example, $N = \lambda z.\Delta\Delta$ is diverging w.r.t. $\beta_v$-reduction, but it is a surface normal form. Let us now incept probability (with the terms $M$ and $N$ as above).
1. Let $R = (\lambda x.M \oplus xx)(\lambda x.M \oplus xx)$. Then $[R] \underset{\mathrm{E}}{\rightrightarrows} [M \oplus R] \underset{\mathrm{E}}{\rightrightarrows} [\frac{1}{2}M, \frac{1}{2}R] \underset{\mathrm{E}}{\rightrightarrows} [\frac{1}{2}I, \frac{1}{2}M \oplus R] \underset{\mathrm{E}}{\rightrightarrows} [\frac{1}{2}I, \frac{1}{4}M, \frac{1}{4}R] \underset{\mathrm{E}}{\rightrightarrows} \cdots$. At the limit, $R$ converges with probability 1 to $I$, as wanted.

$$\frac{}{\langle n : \mathbf{W}(\!|\mathtt{tick}.P|\!)\rangle \underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}} \langle n+1 : \mathbf{W}(\!|P|\!)\rangle} \qquad \frac{M \underset{\mathsf{w}}{\rightarrow}_{\beta_v} M'}{\langle n : M\rangle \underset{\mathsf{w}}{\rightarrow}_{\beta_v} \langle n : M'\rangle} \qquad \frac{M \rightarrow_{\beta_v} M'}{\langle n : M\rangle \rightarrow_{\beta_v} \langle n : M'\rangle}$$

**Figure 5** Payoff reductions $\underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}}, \rightarrow_{\beta_v}, \underset{\mathsf{w}}{\rightarrow}_{\beta_v} \subseteq (\mathbb{N} \times \Lambda_{\mathtt{tick}})$.

$$\frac{M \mapsto_{\beta_v} M'}{\langle 0 : M\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle 0 : M'\rangle} \qquad \frac{M \mapsto_{\mathtt{tick}} M'}{\langle 0 : M\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle 1 : M'\rangle} \qquad \frac{}{\langle 0 : V\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle 0 : V\rangle}$$

$$\frac{\langle 0 : P_1\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle k_1 : P_1\rangle \quad \langle 0 : P_2\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle k_2 : P_2\rangle}{\langle 0 : P_1P_2\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle k_1+k_2 : P_1P_2\rangle} \qquad \frac{n>0 \quad \langle 0 : M\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle k : M'\rangle}{\langle n : M\rangle \underset{/\!/\mathsf{w}}{\rightarrow} \langle n+k : M'\rangle}$$

**Figure 6** Parallel weak reduction in the payoff calculus.

2. The term $S = (\lambda x.N \oplus xx)(\lambda x.N \oplus xx)$ converges to normal form with probability 0. One can easily check that $[S] \underset{\mathsf{E}}{\rightrightarrows}^{\infty} \mathbf{0}$.
3. The term $S' = (\lambda x.(N \oplus I) \oplus xx)(\lambda x.(N \oplus I) \oplus xx)$ converges with probability $\frac{1}{2}$ to the normal form $I$. One can easily check that $[S'] \underset{\mathsf{E}}{\rightrightarrows}^{\infty} \{I^{\frac{1}{2}}\}$.

▶ **Example 33.** One can easily build probabilistic terms with a more interesting behaviour than those in Example 32. First, observe that for $F = \lambda x.I$ (encoding the boolean *false*), we have that $(\lambda z.FF) \rightarrow_{\beta_v} F$. Now let $U = \lambda xy.\big(y \oplus xx(\lambda z.yy)\big)$ and consider the term $UUF$, which converges with probability 1 to $F$. Indeed $[UUF] \underset{\mathsf{E}}{\rightrightarrows}^{\infty} \{F^1\}$. In contrast, surface reduction converges to a distribution over *countably many different surface normal forms*, since each iteration produces a new *snf*: $\frac{1}{2}F, \frac{1}{4}\lambda z.FF, \frac{1}{8}\lambda z.(\lambda z.FF)(\lambda z.FF), \ldots$.

## 6 Asymptotic Normalization: More Case Studies

Our method applies – uniformly – to the other examples in Sect. 1.1. In this section we consider a CbV $\lambda$-calculus extended with an output operator. For the sake of a compact presentation, we take as output not a string, but simply an integer (think of it as a string on a single character). Albeit simple, this case study allows us to illustrate the subtleties related to limits with output calculi, and the use of our method. In a similar way, one can revisit Böhm Trees as the limit of a specific asymptotic strategy – we leave this to Appendix D.2.

**$\lambda$-calculus with output: the payoff calculus.** The payoff $\lambda$-calculus (called cost $\lambda$-calculus in [18,24]) extends the $\lambda$-calculus with a ticking operation. Its intrinsic purpose is to facilitate an intensional analysis of programs, endowing terms with constructs to perform cost analysis.

Let $\Lambda_{\mathtt{tick}}$ denote the set of $\lambda$-terms extended with a unary operator $\mathtt{tick}$. The elements of the payoff calculus are pairs $\mathtt{m} = \langle n : M\rangle$ of a counter $n \in \mathbb{N}$ and a closed term $M \in \Lambda_{\mathtt{tick}}$. Intuitively, the term $\mathtt{tick}(P)$ increments the counter by 1, and continues as $P$. Following [18], in $(\mathbb{N} \times \Lambda_{\mathtt{tick}})$ we define the full reduction $\rightarrow$ and the weak reduction $\underset{\mathsf{w}}{\rightarrow}$ as follows:

$$\rightarrow \;:=\; \rightarrow_{\beta_v} \cup \underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}} \qquad \underset{\mathsf{w}}{\rightarrow} \;:=\; \underset{\mathsf{w}}{\rightarrow}_{\beta_v} \cup \underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}}$$

where $\underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}}, \rightarrow_{\beta_v}, \underset{\mathsf{w}}{\rightarrow}_{\beta_v} \subseteq (\mathbb{N} \times \Lambda_{\mathtt{tick}})$ are given in Fig. 5. Note that weak effectful reduction $\underset{\mathsf{w}}{\rightarrow}_{\mathtt{tick}}$ is the closure under weak context $\mathbf{W}$ of the rule $(\mathtt{tick}.P) \mapsto_{\mathtt{tick}} P$ (effects are only allowed under weak context). Left and right reductions $\underset{\mathsf{l}}{\rightarrow}$ and $\underset{\mathsf{r}}{\rightarrow}$ can be defined similarly.

The pair $((\Lambda_{\texttt{tick}}, \to), \texttt{obs})$ is a QARS where we observe the payoff, *i.e.* $\texttt{obs}\langle n : M\rangle = n$. We now prove (using Thm. 16) that $\underset{\mathsf{w}}{\to} = \underset{\mathsf{w}}{\to}_{\beta_v} \cup \underset{\mathsf{w}}{\to}_{\texttt{tick}}$ is asymptotically complete for $\to$.

▶ **Lemma 34.** *For every pair* $\texttt{m} = \langle n : M\rangle$*,* $\texttt{m} \to^{\infty} \boldsymbol{n}$ *implies* $\texttt{m} \underset{\mathsf{w}}{\to}^{\infty} \boldsymbol{n}$*, because*

- $\mathsf{w}$*-factorization of* $\to$*: if* $\texttt{m} \to^{*} \texttt{n}$ *then* $\texttt{m} \underset{\mathsf{w}}{\to}^{*} \cdot \underset{\neg\mathsf{w}}{\to}^{*} \texttt{n}$*;*
- $\texttt{obs}$*-neutrality : if* $\texttt{m} \underset{\neg\mathsf{w}}{\to} \texttt{m}'$ *then* $\texttt{obs}(\texttt{m}) = \texttt{obs}(\texttt{m}')$*.*

Weak reduction $\underset{\mathsf{w}}{\to}$ however does *not* have a unique limit, as Example 35 below illustrates. An unsatisfactory solution would be to fix a deterministic evaluation order (left or right, as in point 1. below), making the limit easy to predict but also rather *arbitrary*.

▶ **Example 35.** Consider $M = (\Delta\Delta)(\Delta_{\surd}\Delta_{\surd})$, where $\Delta_{\surd} = \lambda x.\texttt{tick}(xx)$, and let $\texttt{m} = \langle 0 : M\rangle$.

1. By fixing left (resp. right) evaluation, $\texttt{Lim}_{\texttt{obs}}(\texttt{m}, \underset{\mathsf{l}}{\to}) = \{0\}$ (resp. $\texttt{Lim}_{\texttt{obs}}(\texttt{m}, \underset{\mathsf{r}}{\to}) = \{\infty\}$).
2. By choosing a redex in unspecified order, we have an uncountable number of $\underset{\mathsf{w}}{\to}$-sequences, leading to $\texttt{Lim}_{\texttt{obs}}(\texttt{m}, \underset{\mathsf{w}}{\to}) = \{0, 1, \dots \infty\} = \mathbb{N}^{\infty}$.

A way out is to proceed somehow similarly to Sect. 5.1. If we examine more closely the set of limits associated with $\underset{\mathsf{w}}{\to}$, we realize that $\texttt{Lim}_{\texttt{obs}}(\texttt{m}, \underset{\mathsf{w}}{\to})$ does have a greatest element. Thus $[\![\texttt{m}]\!]$ can naturally be defined as the best possible payoff from $\texttt{m}$. We prove that parallel reduction $\underset{/\!/\mathsf{w}}{\to}$ (given in Fig. 6) is a (*multistep*) strategy which is guaranteed to compute $\texttt{m}$. Indeed, it is easy to verify that $\underset{/\!/\mathsf{w}}{\to}$ is asymptotically complete for $\underset{\mathsf{w}}{\to}$. By composing with Lemma 34 we have that $\underset{/\!/\mathsf{w}}{\to}$ is asymptotically complete for $\to$ (point 1. below).

▶ **Lemma 36.**

1. Asymptotic Completeness. *If* $\langle k : M\rangle \underset{\mathsf{w}}{\to}^{\infty} \boldsymbol{n}$ *then* $\langle k : M\rangle \underset{/\!/\mathsf{w}}{\to}^{\infty} \boldsymbol{n}'$ *and* $\boldsymbol{n} \leq \boldsymbol{n}'$*. That is,* $\underset{/\!/\mathsf{w}}{\to}$ *is asymptotically complete for* $\underset{\mathsf{w}}{\to}$ *and (by Lemma 34) for* $\to$*.*
2. Unique Limit. *The reduction* $\underset{/\!/\mathsf{w}}{\to}$ *is deterministic.*

Since (by points 1. and 2. in Lemma 36) both conditions of Prop. 14 are verified, we have:

▶ **Theorem 37** (Main, payoff)**.** *Given the QARS* $((\mathbb{N} \times \Lambda_{\texttt{tick}}, \to), \texttt{obs})$*, for each pair* $\texttt{m} = \langle k : M\rangle$*,* $[\![\texttt{m}]\!]$ *is defined, and* $\texttt{m} \underset{/\!/\mathsf{w}}{\to}^{\infty} [\![\texttt{m}]\!]$*. Hence, multistep reduction* $\underset{/\!/\mathsf{w}}{\to}$ *is asymptotically normalizing for* $\underset{\mathsf{w}}{\to}$ *and for* $\to$ *(Remark 15).*

**λ-calculus with outputs.** The calculus in Example 2 can be formalized in a similar way to the payoff calculus. We can define $\texttt{obs}(\langle s : M\rangle) = s$. As already noted, $\underset{\mathsf{w}}{\to}$ is not confluent, and given a pair $\texttt{m}$, the set of limits may contain uncountably many different elements. Still, the reduction has interesting properties, which appear when looking not directly at the string $s$ itself, but at its length $|s|$. This way, one can transfer the results from the payoff calculus.

## 7 Conclusions

We propose a method to study completeness and normalization when the result of computation is *asymptotic*. Our techniques abstract from details specific to the calculus under study – they are therefore of *general* application. The robustness of the method is witnessed by its ability to deal with different settings and *different notions of asymptotic computation*.

The application to probabilistic λ-calculus yields a result of independent interest: a theorem of *asymptotic normalization*, both for CbV and CbN probabilistic λ-calculi. Remarkably, the same definitions and proof techniques apply *uniformly* to both. In the paper we prefer to give the details for the CbV calculus, which is arguably a more natural one in presence of effects.

**Related work.** QARS, proposed in [14] in the setting of probabilistic rewriting, refine Ariola and Blom's ARSI [4]. The techniques in Sect. 3 are an original contribution of this paper. Our Thm. 16 generalizes an ARS technique for finitary normalization (studied in [2, 21, 33]) to asymptotic computation, refining it for arbitrary observations.

The study of reduction strategies in a probabilistic $\lambda$-calculus where the notion of reduction is general – rather than simply fixing a deterministic reduction – started in [16] (CbV and CbN) and [27] (CbN). Asymptotic completeness is there established only for *surface* normal forms (values in closed CbV, hnf's in CbN). Strategies that are complete for *full normal forms* (which we treat and solve here) are more difficult to study than head or weak reduction, especially in the CbV setting. The question of defining such a strategy was left open in [16, Remark 27]. We stress that our technique would also yield a simpler proof of the results in [16, 27], where confluence is used to establish that a greatest limit exists. The (non-trivial) proofs there use properties that are *specific* to probability distributions. The method we propose here avoids technical issues, it is much simpler, and it is general, in that it can be applied to other settings.

Finally, we mention that forgoing confluence and studying uniqueness of normal forms via a complete subreduction is a route already employed in the context of infinitary $\lambda$-calculi [8, 9].

## References


1. Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Inf. Comput.*, 105(2):159–267, 1993. doi:10.1006/inco.1993.1044.
2. Beniamino Accattoli, Claudia Faggian, and Giulio Guerrieri. Factorization and Normalization, Essentially. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019*, volume 11893 of *Lecture Notes in Computer Science*, pages 159–180. Springer, 2019. doi:10.1007/978-3-030-34175-6_9.
3. Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998. doi:10.1017/CBO9780511983504.
4. Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117(1):95–168, 2002. doi:10.1016/S0168-0072(01)00104-X.
5. Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. *Sci. Comput. Program.*, 185, 2020. doi:10.1016/j.scico.2019.102338.
6. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.
7. Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.
8. Henk Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. Comput.*, 207(5):559–582, 2009. doi:10.1016/j.ic.2008.09.003.
9. Alessandro Berarducci and Benedetto Intrigila. Church-Rosser $\lambda$-theories, infinite $\lambda$-calculus and consistency problems. In W. Hodges, M. Hyland, and et. al., editors, *Logic: From Foundations to Applications (European Logic Colloquium)*, pages 33–58. Oxford Sci. Publ., 1996.
10. Gianluca Curzi and Michele Pagani. The Benefit of Being Non-Lazy in Probabilistic $\lambda$-calculus: Applicative Bisimulation is Fully Abstract for Non-Lazy Probabilistic Call-by-Name. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 327–340. ACM, 2020. doi:10.1145/3373718.3394806.
11. Roel C. de Vrijer. Conditional linearization. *Indagationes Mathematicae*, 10(1):145–159, 1999. doi:10.1016/S0019-3577(99)80012-3.
12. Ugo de'Liguoro and Adolfo Piperno. Non Deterministic Extensions of Untyped Lambda-Calculus. *Inf. Comput.*, 122(2):149–177, 1995. doi:10.1006/inco.1995.1145.

**13** Claudia Faggian. Probabilistic Rewriting: Normalization, Termination, and Unique Normal Forms. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019*, volume 131 of *LIPIcs*, pages 19:1–19:25. Schloss Dagstuhl, 2019. `doi:10.4230/LIPIcs.FSCD.2019.19`.

**14** Claudia Faggian. Probabilistic Rewriting and Asymptotic Behaviour: on Termination and Unique Normal Forms. *Log. Methods Comput. Sci., vol. 18, issue 2*, 2022.

**15** Claudia Faggian and Giulio Guerrieri. Strategies for Asymptotic Normalization (long version). *CoRR*, 2022. `arXiv:2204.08772`.

**16** Claudia Faggian and Simona Ronchi Della Rocca. Lambda Calculus and Probabilistic Computation. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019*, pages 1–13. IEEE, 2019. `doi:10.1109/LICS.2019.8785699`.

**17** Francesco Gavazzo. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects*. PhD thesis, Università di Bologna, Italy, 2019. URL: `https://tel.archives-ouvertes.fr/tel-02386201`.

**18** Francesco Gavazzo and Claudia Faggian. A Relational Theory of Monadic Rewriting Systems, Part I. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–14. IEEE, 2021. `doi:10.1109/LICS52264.2021.9470633`.

**19** Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, pages 235–246. ACM, 2002. `doi:10.1145/581478.581501`.

**20** J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

**21** Nao Hirokawa, Aart Middeldorp, and Georg Moser. Leftmost Outermost Revisited. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015*, volume 36 of *LIPIcs*, pages 209–222. Schloss Dagstuhl, 2015. `doi:10.4230/LIPIcs.RTA.2015.209`.

**22** J. Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. Comput.*, 119(1):18–38, 1995. `doi:10.1006/inco.1995.1075`.

**23** J. Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997. `doi:10.1016/S0304-3975(96)00171-5`.

**24** Ugo Dal Lago and Francesco Gavazzo. Effectful Normal Form Bisimulation. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 263–292. Springer, 2019. `doi:10.1007/978-3-030-17184-1_10`.

**25** Ugo Dal Lago and Margherita Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO Theor. Informatics Appl.*, 46(3):413–450, 2012. `doi:10.1051/ita/2012012`.

**26** Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990. URL: `http://gallium.inria.fr/~xleroy/publi/ZINC.pdf`.

**27** Thomas Leventis. A deterministic rewrite system for the probabilistic $\lambda$-calculus. *Mathematical Structures in Computer Science*, 29(10):1479–1512, 2019. `doi:10.1017/S0960129519000045`.

**28** Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda calcul*. PhD thesis, Université Paris 7, 1978. URL: `http://pauillac.inria.fr/~levy/pubs/74phd-cycle3.pdf`.

**29** Maxwell H. A. Newman. On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2), 1942.

**30** 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`.

**31** Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

32 Vincent van Oostrom. Random Descent. In *Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2007. doi:10.1007/978-3-540-73449-9_24.

33 Vincent van Oostrom and Yoshihito Toyama. Normalisation by Random Descent. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016*, volume 52 of *LIPIcs*, pages 32:1–32:18. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.FSCD.2016.32.

# APPENDIX

We include some proofs and details that have been omitted in the article.

**One more example.** $\mathtt{Lim}_{\mathtt{obs}}(t, \to)$ may have a lub but not a maximum – similarly to $\mathbb{N}$.

▶ **Example 38** (Sect. 2, QARS). We revisit Example 11, allowing full reduction $\to_{\beta_v}$. Let $\mathtt{obs}_p(\langle s : M\rangle) = s$ if $M \in \mathcal{V}$, $\bot$ otherwise. The pair $\mathtt{m} = \langle \epsilon : (\lambda z.I)(\lambda z.\Delta_0\Delta_0)\rangle$ has countably many limits, but not a greatest one, because all strings in $\mathtt{Lim}_{\mathtt{obs}_p}(\mathtt{m}, \to_{\beta_v})$ are finite.

**Surface reduction.** Everywhere in the appendix, we fix surface reduction to be as follows.

- CbN ($\mathsf{b} = \beta$): $\mathsf{s} = \mathsf{h}$ (the contextual closure of $\mathbf{H}$).
- CbV ($\mathsf{b} = \beta_v$): $\mathsf{s} = \mathsf{w}$ (the contextual closure of $\mathbf{W}$).

## A Properties of surface normal forms

We will use extensively the following easy fact.

▶ **Lemma 39** (Surface normal forms). *$M$ is $\mathsf{w}$-normal (resp. $\mathsf{h}$-normal) if there is no redex $R$ such that $M = \mathbf{W}(\!|R|\!)$ (resp. $M = \mathbf{H}(\!|R|\!)$).*

1. *CbV. Assume $M \underset{\neg\mathsf{w}}{\to}{}_{\beta_v} M'$. $M$ is $\mathsf{w}$-normal $\Leftrightarrow$ $M'$ is $\mathsf{w}$-normal.*
2. *CbN. Assume $M \underset{\neg\mathsf{h}}{\to}{}_{\beta} M'$. $M$ is $\mathsf{h}$-normal $\Leftrightarrow$ $M'$ is $\mathsf{h}$-normal.*

## B Sect. 4.2: properties of unbiased reduction

The properties of $\underset{\mathrm{U}}{\to}$ ( Prop. 21, Prop. 22, Thm. 23, and those stted here) are proved in [15].

▶ **Lemma 40** (Normal forms). *If $U \underset{\neg\mathrm{U}}{\to}{}_{\mathsf{b}} N$, then $N$ is not $\mathsf{b}$-normal.*

### B.1 A parallel variant of unbiased reduction

Given $(\Lambda, \to_{\mathsf{b}})$ and $\underset{\mathsf{s}}{\to}$ as in Def. 20, a parallel version $\underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}}$ is easily defined. The idea here is that once a term is $\underset{\mathsf{s}}{\to}$-normal, iteration of the reduction process can be performed in any arbitrary order, or in *parallel*. Recall that here $(\mathsf{b}, \mathsf{s}) \in \{(\beta, \mathsf{h}), (\beta_v, \mathsf{w})\}$.

1. If $M \underset{\mathsf{s}}{\to} M'$ then $M \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} M'$ ($M$ is not $\mathsf{s}$-normal);
2. If $M \not\to$ then $M \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} M$ ($M$ is $\to$-normal);
3. Otherwise:

$$\frac{P \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} P'}{M := \lambda x.P \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} \lambda x.P'} \qquad \frac{P_1 \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} P_1' \quad P_2 \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} P_2'}{M := P_1P_2 \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} P_1'P_2'} \qquad \frac{(P_i \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} P_i')_{1\le i\le k}}{M := \mathbf{o}(P_1, \dots, P_k) \underset{/\!/\mathrm{U}}{\to}{}_{\mathsf{b}} \mathbf{o}(P_1', \dots, P_k')}$$

Rule 2. makes the relation reflexive on normal forms and *only* on normal forms – this is a harmless shortcut in order to give a compact and neat formulation.

The (multistep) reduction $\underset{//\mathrm{U}}{\rightarrow}_{\mathtt{b}}$ is guaranteed to reach the $\rightarrow_{\mathtt{b}}$-*nf*, if any exists.

▶ **Lemma 41.** *Let* $\mathtt{b} \in \{\beta, \beta_v\}$

1. *If* $M \underset{//\mathrm{U}}{\rightarrow}_{\mathtt{b}} N$ *then* $M \underset{\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N$*. Therefore,* $M \underset{//\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N$ *implies* $M \underset{\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N$*.*
2. *If* $M \underset{\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N$ *then there exists* $N'$ *such that* $M \underset{//\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N'$ *and* $N \underset{\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N'$*.*

▶ **Corollary 42** ($\mathcal{N}$-completeness)**.** *Let* $\mathtt{b} \in \{\beta, \beta_v\}$ *and* $N$ *be* $\rightarrow_{\mathtt{b}}$*-normal.* $M \rightarrow_{\mathtt{b}}{}^* N$ *if and only if* $M \underset{//\mathrm{U}}{\rightarrow}_{\mathtt{b}}{}^* N$*.*

## C Proofs of Sect. 5.1: Asymptotic Normalization for $\Lambda_{\oplus}^{\mathsf{cbv}}$

Notice also that the definition of the reductions $\underset{\mathrm{E}}{\rightarrow}, \underset{\mathrm{E}}{\Rightarrow}$ can be given in the same way also in CbN, by replacing $\beta_v$ with $\beta$ (surface steps are here head steps).

**Properties.** We freely use the following fact.

▶ **Fact 43.** $[M] \Rightarrow_{\beta_v} [M']$ *iff* $M \rightarrow_{\beta_v}{}^{=} M'$ *, where*

- *on the l.h.s. we have* $(\mathcal{M}(\Lambda_\oplus), \Rightarrow)$*, and*
- *on the r.h.s. the CbV* $\lambda$*-calculus* $(\Lambda_\oplus, \rightarrow_{\beta_v})$*, as defined in Sect. 4.1.*

**Factorization and Neutrality.** Recall that $\mathcal{S}$ denotes the set of the **surface normal forms** of $\rightarrow = (\rightarrow_{\beta_v} \cup \rightarrow_\oplus)$. $M \in \Lambda_\oplus$ is $\mathsf{s}$-normal if $M \not\underset{\mathsf{s}}{\rightarrow}$. That is $M \not\rightarrow_\oplus$ and $M \not\underset{\mathsf{s}}{\rightarrow}_{\beta_v}$.

▶ **Lemma 44** (*snf* propagation)**.** *If* $M$ *is* $\mathsf{s}$*-normal and* $M \rightarrow M'$*, then* $M'$ *is* $\mathsf{s}$*-normal.*

▶ **Proposition 45** (E-Factorization of $\Rightarrow$.)**.** *In* $\Lambda_{\oplus}^{\mathsf{cbv}}$*:* $\mathtt{m} \Rightarrow^* \mathtt{n}$ *implies* $\mathtt{m} \underset{\mathrm{E}}{\Rightarrow}{}^* \cdot \underset{\neg\mathrm{E}}{\Rightarrow}{}^* \mathtt{n}$

**Proof.** In the proof, we use freely Fact 43. By surface factorization of $\Rightarrow$ (proved in [16]), $\mathtt{m} \Rightarrow^* \mathtt{n}$ implies $\mathtt{m} \underset{\mathsf{s}}{\Rightarrow}{}^* \mathtt{t} \underset{\neg\mathsf{s}}{\Rightarrow}{}^* \mathtt{n}$ for some $\mathtt{t}$. From this we have:

- $\mathtt{m} \underset{\mathrm{E}}{\Rightarrow}{}^* \mathtt{t}$. Because if $M \underset{\mathsf{s}}{\rightarrow} \mathtt{r}$ then also $M \underset{\mathrm{E}}{\rightarrow} \mathtt{r}$.
- $\mathtt{t} \underset{\neg\mathsf{s}}{\Rightarrow}_{\beta_v}{}^* \mathtt{n}$. Because if $M \underset{\neg\mathsf{s}}{\rightarrow} \mathtt{r}$ then necessarily $M \underset{\neg\mathsf{s}}{\rightarrow}_{\beta_v} \mathtt{r}$. Moreover, $\mathtt{r} = [M']$.

Let $\mathtt{t} = [\dots p_i T_i \dots]_{i\in I}$. Then necessarily, $\mathtt{n} = [\dots p_i N_i \dots]_{i \in I}$ and $[T_i] \underset{\neg\mathsf{s}}{\Rightarrow}_{\beta_v}{}^* [N_i]$ and so also $T_i \underset{\neg\mathsf{s}}{\rightarrow}_{\beta_v}{}^* N_i$. For each $T_i$, we examine if $T_i$ is $\mathsf{s}$-normal or not ($\mathcal{S}$ being the set of *snf*'s).

1. $T_i \in \mathcal{S}$. By $T_i \underset{\neg\mathsf{s}}{\rightarrow}_{\beta_v}{}^* N_i$ and U-factorization of $\rightarrow_{\beta_v}$ (Prop. 21), $T_i \underset{\mathrm{U}}{\rightarrow}_{\beta_v}{}^* U_i \underset{\neg\mathrm{U}}{\rightarrow}_{\beta_v}{}^* N_i$. By Lemma 44, each term in the sequence $T_i \underset{\mathrm{U}}{\rightarrow}_{\beta_v}{}^* U_i \underset{\neg\mathrm{U}}{\rightarrow}_{\beta_v}{}^* N_i$ is $\mathsf{s}$-normal. Hence, by Def. 24 (since only the second rule can apply), we conclude that $[T_i] \underset{\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* [U_i] \underset{\neg\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* [N_i]$.
2. $T_i \notin \mathcal{S}$. By Lemma 39, each term in the sequence $T_i \underset{\neg\mathsf{s}}{\rightarrow}_{\beta_v}{}^* N_i$ is not $\mathsf{s}$-normal. By definition of $\underset{\mathrm{E}}{\rightarrow}$ (since only the first rule can apply) we conclude that $[T_i] \underset{\neg\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* [N_i]$.

Let us partition $\mathtt{t}$ into two multi-distributions, collecting in $\mathtt{t}_1$ the terms of case 1. and in $\mathtt{t}_2$ the terms of case 2. We partition $\mathtt{n}$ so that $\mathtt{t}_1 \underset{\neg\mathsf{s}}{\Rightarrow}{}^* \mathtt{n}_1$ and $\mathtt{t}_2 \underset{\neg\mathsf{s}}{\Rightarrow}{}^* \mathtt{n}_2$. We have $\mathtt{t}_1 \underset{\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* \mathtt{u} \underset{\neg\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* \mathtt{n}_1$ and $\mathtt{t}_2 \underset{\neg\mathrm{E}}{\Rightarrow}_{\beta_v}{}^* \mathtt{n}_2$. Therefore $\mathtt{m} \underset{\mathrm{E}}{\Rightarrow}{}^* (\mathtt{t}_1{+}\mathtt{t}_2) \underset{\mathrm{E}}{\Rightarrow}{}^* (\mathtt{u}{+}\mathtt{t}_2) \underset{\neg\mathrm{E}}{\Rightarrow}{}^* (\mathtt{n}_1{+}\mathtt{n}_2) = \mathtt{n}$. which proves the claim. ◀

▶ **Proposition 46** (neutrality)**.** *If* $\mathtt{m} \underset{\neg E}{\Rightarrow} \mathtt{n}$ *then* $\mathtt{obs}_{\mathcal{N}}(\mathtt{m}) = \mathtt{obs}_{\mathcal{N}}(\mathtt{n})$*.*

**Proof.** Consequence of the fact that if $U \underset{\neg\mathrm{U}}{\rightarrow}_{\beta_v} N$, then $N$ is not $\beta_v$-normal (Lemma 40). Indeed $\underset{\neg\mathrm{E}}{\rightarrow} \subseteq \underset{\neg\mathsf{s}}{\rightarrow}$ and so $M \underset{\neg\mathrm{E}}{\rightarrow} \mathtt{r}$ *iff* ( $M \underset{\neg\mathrm{U}}{\rightarrow}_{\beta_v} M'$ with $\mathtt{r} = [M']$). ◀

**Diamonds.** Prop. 27 (the relation $\underset{E}{\rightrightarrows}$ is $\mathtt{obs}_{\mathcal{N}}$-diamond) follows from the following key lemma. Notice that Point (2.) implies that $\mathtt{obs}_{\mathcal{N}}(\mathtt{m}_1) = \mathtt{obs}_{\mathcal{N}}(\mathtt{m}_2)$.

▶ **Lemma 47** (Pointed Diamond). *Let $\alpha, \gamma \in \{\beta_v, \oplus\}$. Assume $M$ has two distinct redexes, such that $M \underset{E}{\rightrightarrows}_\alpha \mathtt{m}_1$ and $M \underset{E}{\rightrightarrows}_\gamma \mathtt{m}_2$. Then*

1. *exists $\mathtt{t}$ such that $\mathtt{m}_1 \underset{E}{\rightrightarrows}_\gamma \mathtt{t}$ and $\mathtt{m}_2 \underset{E}{\rightrightarrows}_\alpha \mathtt{t}$.*
2. *Moreover, no $M_i$ in $\mathtt{m}_1 = [p_i M_i]_i$ and no $M_j$ in $\mathtt{m}_2 = [q_j M_j]_j$ is $\to$normal.*

**Proof.**

- If $M$ is $\mathsf{s}$-normal, then by definition of $\underset{E}{\rightrightarrows}$, $M \underset{U}{\rightrightarrows}_{\beta_v} \mathtt{m}_1$ and $M \underset{U}{\rightrightarrows}_{\beta_v} \mathtt{m}_2$, and we conclude by using Fact 43 and Prop. 22, point 1.
- If $M$ is $\mathsf{s}$-reducible, then by definition of $\underset{E}{\rightrightarrows}$, $M \underset{\mathsf{s}}{\rightrightarrows}_{\beta_v} \mathtt{m}_1$ and $M \underset{\mathsf{s}}{\rightrightarrows}_{\beta_v} \mathtt{m}_2$. We easily conclude by case analysis. ◀

▶ Remark 48 ($\underset{E}{\rightrightarrows} \neq \underset{U}{\rightrightarrows}_{\mathtt{b}} \cup \to_\oplus$). It is useful to notice that $\underset{E}{\rightrightarrows} \neq \underset{U}{\rightrightarrows}_{\beta_v} \cup \to_\oplus$. Such a relation is neither diamond nor confluent.

- The lifting of $\underset{U}{\rightrightarrows}_{\beta_v} \cup \to_\oplus$ is neither diamond nor confluent. Consider $(\Delta \oplus \Delta\Delta)(\lambda z.Iz)$. Then $\mathtt{m}_1 = [\frac{1}{2}\Delta(\lambda z.Iz), \frac{1}{2}(\Delta\Delta)(\lambda z.\underline{Iz})] \leftarrow_\oplus (\Delta \oplus \Delta\Delta)(\lambda z.Iz) \underset{U}{\rightrightarrows}_{\beta_v} [(\Delta \oplus \Delta\Delta)(\lambda z.z)] = \mathtt{m}_2$. The elements $\mathtt{m}_1$ and $\mathtt{m}_2$ cannot join, because no $\underset{U}{\rightrightarrows}_{\beta_v}$-step can fire the underlined $(Iz)$.
- Similarly in CbN, for the lifting of $\underset{U}{\rightrightarrows}_\beta \cup \to_\oplus$. Consider $(\Delta \oplus \Delta\Delta)(x(Iz))$.

# D Details for Sect. 6: more case studies

## D.1 Asymptotic Normalization for a calculus with outputs

**Proof of Lemma 34.** The $\mathsf{w}$-factorization of $\to$ is proved in [18], where it is called surface factorization, and proved in general for all CbV monadic calculi, including the payoff calculus which we discuss here. $\mathtt{obs}$-neutrality is straightforward to verify, by case analysis.

## D.2 Asymptotic Normalization and Böhm Trees

We show that the Böhm Tree of a term $M$ is the (unique) limit of an asymptotically normalizing strategy, *i.e.* the limit of a *single* reduction sequence.

**Böhm Trees and Partial Normal Forms.** Following [3], the Böhm Tree of a term $M$ is (the downward closure of) the set of the partial normal forms of all reducts of $M$.

▶ **Definition 49** (Partial Normal Forms and Böhm Trees ). *The set $\mathcal{N}_\omega$ of* ***partial normal forms*** *is defined as follows;*

$$\frac{}{\Omega \in \mathcal{N}_\omega} \qquad \frac{A_1 \in \mathcal{N}_\omega \quad \dots \quad A_n \in \mathcal{N}_\omega}{\lambda x_1 \dots x_n.x A_1 \dots A_n \in \mathcal{N}_\omega}$$

*$\mathcal{N}_\omega$ is a subset of the set of partial $\lambda$-terms, defined by $P := \Omega \mid x \mid PP \mid \lambda x.P$, and inherits its* ***order*** *$\leq$, which is generated by the following rules:*

$$\frac{}{\Omega \leq P} \qquad \frac{P_1 \leq P_1' \quad P_2 \leq P_2'}{P_1 P_2 \leq P_1' P_2'} \qquad \frac{P \leq P'}{\lambda x.P \leq \lambda x.P'}$$

*The elements of the ideal completion $\mathcal{N}_\omega^\infty$ of $\mathcal{N}_\omega$ are called* ***Böhm Trees****. Precisely:*

1. *The function* $\omega : \Lambda \to \mathcal{N}_\omega$ *associates to each term* $M \in \Lambda$ *its* partial normal form $\omega(M)$*:*

$$\omega(M) = \begin{cases} \Omega & \text{if } M \notin \mathcal{H} \\ \lambda\vec{x}.x\omega(M_1)\dots\omega(M_p) & \text{if } M = \lambda\vec{x}.xM_1...M_p \end{cases}$$

2. *The* ***Böhm Tree of*** $M$ *is defined as below For a set* $\mathcal{S}$*,* $\downarrow\mathcal{S} = \{Q \in \mathcal{N}_\omega \mid Q \leq S \in \mathcal{S}\}$*.*

$$\mathtt{BT(M)} := \bigcup_{M \to^* N} \downarrow\{\omega(N)\} = \downarrow\{\omega(N) \mid M \to^* N\}$$

The following property is standard and easy-to-check (see [3, Lemma 2.3.2].)

▶ **Lemma 50.** *Let* $M, M' \in \Lambda$*. If* $M \to_\beta M'$ *then* $\omega(M) \leq \omega(M')$*.*

Lemma 50 guarantees that $((\Lambda, \to_\beta), \mathtt{obs})$ is a QARS where $\mathtt{obs}\colon \Lambda \to \mathcal{N}_\omega^\infty$ is defined as $\mathtt{obs}(M) = \downarrow\{\omega(M)\}$.

**Asymptotic Normalization.** Let us define $\mathtt{obs} : \Lambda \to \mathcal{N}_\omega^\infty$ as $\mathtt{obs}(M) = \downarrow\{\omega(M)\}$. It is easily checked that $\big((\Lambda, \to_\beta), \mathtt{obs}\big)$ is a QARS. We show that the Böhm Tree of a term $M$ can be obtained by asymptotic normalization, as the limit a $\underset{/\!/\mathrm{U}}{\to}_\beta$ reduction sequence, which is an $\mathtt{obs}$-normalizing strategy for $\to_\beta$ (Thm. 54).

The $\mathtt{obs}$-limit of a reduction sequence $\langle M_n \rangle_n$ is then $\sup_i\{\mathtt{obs}(M_i)\} = \bigcup_i \downarrow\{\omega(M_i)\}$. $\mathtt{BT(M)}$ is clearly the sup of the set $\mathtt{Lim}_{\mathtt{obs}}(M, \to_\beta)$. We show that $\mathtt{BT(M)}$ *belongs* to that set, by proving that $\mathtt{Lim}_{\mathtt{obs}}(M, \to_\beta)$ has a greatest element $[\![M]\!]$; this necessarily is $\mathtt{BT(M)}$.

We first show that $\underset{\mathrm{U}}{\to}_\beta$ is $\mathtt{obs}$-complete for $\to_\beta$ (Point 1 in Prop. 52 below). Reduction $\underset{\mathrm{U}}{\to}_\beta$ is not $\mathtt{obs}$-normalizing for $\to_\beta$ (for example, it admits the sequence $x(\Delta\Delta)(Iz) \underset{\mathrm{U}}{\to} x(\Delta\Delta)(Iz) \underset{\mathrm{U}}{\to} \dots$) but its parallel version $\underset{/\!/\mathrm{U}}{\to}$ (Appendix B.1) is.

We proceed similarly to Sect. 5.1 (think $\underset{\mathrm{E}}{\Rrightarrow}$ vs $\underset{\mathrm{E}}{\rightrightarrows}$). We consider the reduction $\underset{/\!/\mathrm{U}}{\to}$ (the explicit definition is in Appendix B.1) which has $\mathtt{obs}$-Random Descent (trivially) and is asymptotically complete for $\underset{\mathrm{U}}{\to}$ (Point 2 in Prop. 52 below), and so for $\to_\beta$.

▶ **Lemma 51.** *If* $M \underset{\neg\mathrm{U}}{\to}_\beta M'$ *then* $\omega(M) = \omega(M')$*.*

**Proof.** First, observe that $M \underset{\neg\mathsf{h}}{\to}_\beta M'$, because $\underset{\neg\mathrm{U}}{\to}_\beta \subseteq \underset{\neg\mathsf{h}}{\to}_\beta$.

- If $M$ is not $\mathsf{h}$-normal ($M \notin \mathcal{H}$), neither is $M'$, by (Lemma 39). Therefore, $\omega(M) = \Omega = \omega(M')$.
- Otherwise, $M$ is $\mathsf{h}$-normal, that is, $M = \lambda\vec{x}.xM_1 \dots M_p$. As $M \underset{\neg\mathsf{h}}{\to}_\beta M'$, necessarily $M' = \lambda\vec{x}.xM_1 \dots M_i' \dots M_p$ (which is head normal) and $M_i \to_\beta M_i'$ for some $1 \leq i \leq p$. It is impossible that $M_i \underset{\mathrm{U}}{\to}_\beta M_i'$, otherwise $M \underset{\mathrm{U}}{\to}_\beta M'$ according to the definition of $\underset{\mathrm{U}}{\to}_\beta$ (Def. 20). Therefore, $M_i \underset{\neg\mathrm{U}}{\to}_\beta M_i'$ and so, by *i.h.*, $\omega(M_i) = \omega(M_i')$. Thus, $\omega(M) = \lambda\vec{x}.x\omega(M_1) \dots \omega(M_i) \dots \omega(M_p) = \lambda\vec{x}.x\omega(M_1) \dots \omega(M_i') \dots \omega(M_p) = \omega(M')$.

▶ **Proposition 52** (Asymptotic Completeness)**.**

1. $M \to_\beta{}^\infty \mathtt{r}$ *implies* $M \underset{\mathrm{U}}{\to}{}^\infty \mathtt{r}$*, because*
   - U*-Factorization of* $\to_\beta$*:* $M \to_\beta{}^* N$ *implies* $M \underset{\mathrm{U}}{\to}{}^* \cdot \underset{\neg\mathrm{U}}{\to}{}^* N$
   - $\mathtt{obs}$*-neutrality :* $M \underset{\neg\mathrm{U}}{\to} M'$ *then* $\mathtt{obs}M = \mathtt{obs}M'$*.*
2. $M \underset{\mathrm{U}}{\to}{}^\infty \mathtt{r}$ *implies* $(M \underset{/\!/\mathrm{U}}{\to}{}^\infty \mathtt{s}$ *and* $\mathtt{r} \leq \mathtt{s})$ ◀

**Proof.**

1. Factorization is that for $\underset{\mathrm{U}}{\to}_\beta$ (details of the proof are in [15]). obs-neutrality is immediate consequence of Lemma 51.
2. It follows by Lemma 41, and the fact that obs is monotonic. ◀

▶ Remark 53 (Unique Limit). If we take for head reduction the standard one (as in [7]), then $\underset{/\!/\mathrm{U}}{\to}$ is deterministic. Otherwise, if we take $\underset{\mathsf{h}}{\to}_\beta$ as defined in Sect. 4.1, it is easily verified that $\underset{/\!/\mathrm{U}}{\to}$ has the obs-diamond property.

$\underset{/\!/\mathrm{U}}{\to}$ is obs-complete for $\to_\beta$ (by Points 1. and 2.). Hence we conclude by Thm. 16:

▶ **Theorem 54** (Main, Böhm Trees). *$\underset{/\!/\mathrm{U}}{\to}$ is a (multi-step)* obs*-normalizing strategy for $\to_\beta$, and $M \underset{/\!/\mathrm{U}}{\to}^{\infty}$* $\mathtt{BT(M)}$.

# Solvability for Generalized Applications

**Delia Kesner**
Université de Paris, CNRS, IRIF, France
Institut Universitaire de France, France

**Loïc Peyrot**
Université de Paris, CNRS, IRIF, France

## Abstract

Solvability is a key notion in the theory of call-by-name λ-calculus, used in particular to identify meaningful terms. However, adapting this notion to other call-by-name calculi, or extending it to different models of computation – such as call-by-value – , is not straightforward. In this paper, we study solvability for call-by-name and call-by-value λ-calculi with generalized applications, both variants inspired from von Plato's natural deduction with generalized elimination rules. We develop an operational as well as a logical theory of solvability for each of them. The operational characterization relies on a notion of solvable reduction for generalized applications, and the logical characterization is given in terms of typability in an appropriate non-idempotent intersection type system. Finally, we show that solvability in generalized applications and solvability in the λ-calculus are equivalent notions.

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

**Keywords and phrases** Lambda-calculus, Generalized applications, Solvability, CBN/CBV, Quantitative types

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.18

## 1 Introduction

The λ-calculus with generalized applications $\Lambda J$ [24, 25] can be seen as a Curry-Howard interpretation of natural deduction with generalized elimination rules, a system developed in parallel by Tenant [34] and von Plato [35]. Like the usual λ-calculus, $\Lambda J$ is a call-by-name (CBN) calculus: the arguments of a function are substituted in its body without any other prior computation. In the call-by-value (CBV) paradigm instead, the arguments are always reduced before substitution [33]. CBV parameter passing is a choice made by many functional programming languages such as OCaml, Scheme or Coq. Very recently, a CBV variant of $\Lambda J$, called $\Lambda J_v$, was introduced by Espírito Santo [16]. It is surprisingly simple, implements general *strong* reduction (inside abstractions), and is very close to the CBN formalism $\Lambda J$: they both share the same notion of reducible expression (*a.k.a. redex*), so that *every* function application is always reducible, because only the underlying notion of substitution differs.

The study of λ-calculi with generalized applications is of special interest for the understanding of the semantics of programming language. They give a fresh look at applications, and also bear similarities with explicit substitutions [1, 27] and the sequent calculus [19]. However, while many previous works [15, 17] were dedicated to the proof-theory underlying different calculi with generalized applications, very few semantical properties have been studied so far in this framework. In this paper, we show that existing tools and techniques for *solvability* can be adapted to this more general model of computation.

Solvability is used to identify *meaningful* terms. Indeed, all solvable terms progressively unveil a stable structure along the reduction process: this gives a step-by-step partial result that is later integrated into the definitive structure of the fully normalized term. Thus, the set of solvable terms contains all normalizable terms, but also a strict subset of the divergent

© Delia Kesner and Loïc Peyrot;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 18; pp. 18:1–18:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ones. On the contrary, if a term containing an unsolvable subterm $u$ converges to a result, then $u$ can be replaced by any other term, still giving the same result and thus justifying the designation of unsolvable as *meaningless* (Genericity Lemma [8]). In semantical models of the λ-calculus, equating all unsolvable terms (*e.g.* to bottom) turns out to be consistent, but this is not the case if one equates in addition terms which are not normalizing [36].

Whilst being an important semantical property, solvability also has a very elegant operational theory. A solvable term may reduce to any other term when closed by abstractions and applied to a suitable sequence of arguments (*i.e.* plugged inside a *head context*). In the CBN λ-calculus, a term $t$ is solvable *iff* $t$ has a *head normal form iff* $t$ *head*-normalizes [36]. But because of the different normalization behaviors of CBN/CBV, their corresponding notions of solvability do not perfectly coincide. In fact, it is only recently that a characterization of CBV solvability making use of some proper notion of CBV reduction has been achieved [7, 12].

In this paper, we give an *operational* characterization of solvability for CBN/CBV generalized applications in terms of an appropriate reduction relation for each case. Indeed, we define a notion of *solvable reduction* for CBN (resp. CBV) for which all and only CBN (resp. CBV) solvable terms normalize. We take advantage of the similarity between CBN and CBV with generalized applications (given by the fact that they both share the same notion of redex, in contrast to the λ-calculus case) to highlight the divergences of the two kinds of solvability. On the one hand, our results show the robustness of the operational theory of solvability. On the other hand, our study of solvability is a step forward in the understanding of programming language semantics based on generalized applications.

In addition to the previously mentioned operational characterizations, we also provide *logical* characterizations of CBN/CBV solvability as typability in *quantitative* (*a.k.a.* non-idempotent intersection) *type systems*. More precisely, we deduce an equivalence between (1) being normalizable for the newly defined solvable reduction relations, (2) being (quantitatively) typable and (3) being solvable. In the λ-calculus, such a result was already well-known for CBN (see [11] for a survey), and it has recently been obtained [5] for CBV, serving as the starting point for our CBV characterization. We will then also prove equivalence between solvability in the CBN/CBV λ-calculus and solvability in CBN/CBV generalized applications by simply reasoning on their associated quantitative type systems. A further advantage of quantitative types is that they enable simple combinatorial proofs of normalization. They also have strong ties with denotational semantics, as it is generally possible to derive a relational model from a quantitative type system.

The adaptation of CBN/CBV solvability in our framework is not trivial, because of the particular syntactical structure of generalized applications which facilitates the existence of *blocked* redexes. To this end, the original calculi with generalized applications $\Lambda J$ and $\Lambda J_v$ use a permutative/commutative conversion $\pi$ on top of the primary rule $\beta$ for CBN and $\beta_{\mathsf{v}}$ for CBV, respectively. This rule is needed to unblock these stuck redexes. Instead, we work with *distant* variants ($\lambda J$ for CBN [18], written here $\lambda J_n$, and a novel distant variant $\lambda J_v$ for CBV). Thus, we integrate the permutations into the primary rules, in the spirit of calculi with explicit substitutions at a distance [6]. In this way, the reduction rules focus on the computational behavior of the language and not on syntactical accidents. In particular, only primary reductions (and not permutations) create divergence, and thus unsolvability. This choice also reflects the logical model in a better way: in our framework, quantitative type systems are neutral to permutations rules (specifically: the size of derivations does not change), as quantitativity is only related to the computational rules. Yet, we show that our results also apply to the original calculi $\Lambda J$ and $\Lambda J_v$.

To summarize, our main contributions are the following:

- We define appropriate notions of head contexts and solvable reductions in calculi with generalized applications and distance.
- We show operational characterizations of CBN/CBV solvability in this setting.
- We also provide (logical) quantitative characterizations of these notions of solvability.
- We derive characterizations of solvability for the original calculi with generalized applications (without distance).
- We show that our definition of solvability corresponds to the one of the λ-calculus.

### Plan of the paper

In Sec. 2 we present the distant calculi with generalized applications for both CBN and CBV. We then introduce their respective notions of solvability in Sec. 3. We give operational and logical characterizations of CBN (resp. CBV) solvability in Sec. 4 (resp. Sec. 5). In Sec. 6 we extend our results to the original (non-distant) calculi $\Lambda J$ and $\Lambda J_v$ and we relate these results to solvability in the λ-calculus. Related works and conclusions are detailed in Sec. 7.

## 2 The Distant Calculi with Generalized Applications

In this section we define the syntax and operational semantics of two distant λ-calculi with generalized applications for call-by-name and call-by-value, noted $\lambda J_n$ and $\lambda J_v$ respectively.

### 2.1 Syntax

Given a countably infinite set $\mathcal{X}$ of variables $x, y, z \dots$, the set of terms generated by the following grammar is called $\mathtt{T}_J$.

$$(\textbf{Values})\ v ::= x \in \mathcal{X} \mid \lambda x.t \qquad (\textbf{Terms})\ t, u, r, s ::= v \mid t(u, x.r)$$

From now on, values are denoted by $v$ and arbitrary terms by $t$, $u$, $r$ or $s$. The term $x$ is a **variable**, $\lambda x.t$ is an **abstraction** and $t(u, x.r)$ is a **generalized application**, whose part $x.r$ – that is not a subterm itself – is called a **continuation**. A generalized application can be seen as a let-binding/explicit substitution under the (informal) translation of $t(u, x.r)$ to $\mathtt{let}\ x = tu\ \mathtt{in}\ r$. **Free** and **bound** variables of terms are defined as expected, in particular, $\mathrm{fv}(\lambda x.r) := \mathrm{fv}(r) \setminus \{x\}$ and $\mathrm{fv}(t(u, x.r)) := \mathrm{fv}(t) \cup \mathrm{fv}(u) \cup (\mathrm{fv}(r) \setminus \{x\})$, so that both $\lambda x.r$ and $t(u, x.r)$ bind the variable $x$ in the term $r$. We work modulo $\alpha$-conversion, so that bound variables can be renamed when necessary. A generalized application $t(u, x.r)$ is said to be **non-relevant** if $x \notin \mathrm{fv}(r)$.

We set some special terms that will be used in forthcoming examples and definitions: $\mathtt{I} := \lambda z.z$, $\delta := \lambda x.x(x, z.z)$, $\Omega := \delta(\delta, z.z)$, as well as a family of projection terms $\mathbf{o}^n := \lambda x_n \dots \lambda x_0.x_0$ parametrized by a natural number $n$.

**Contexts** ($\mathtt{C}$) are terms with one occurrence of the hole $\Diamond$, and **distant contexts** ($\mathtt{D}$) are special contexts used to define the operational semantics of the calculi:

$$\mathtt{C} ::= \Diamond \mid \lambda x.\mathtt{C} \mid \mathtt{C}(u, x.r) \mid t(\mathtt{C}, x.r) \mid t(u, x.\mathtt{C}) \qquad \mathtt{D} ::= \Diamond \mid t(u, x.\mathtt{D})$$

The term $\mathtt{C}\langle t\rangle$ is obtained by replacing $\Diamond$ in the context $\mathtt{C}$ by the term $t$, so that capture of variables may eventually occur. When the free variables of $t$ are not captured by $\mathtt{C}$, we may write $\mathtt{C}\langle\langle t\rangle\rangle$ instead of $\mathtt{C}\langle t\rangle$. Notice that every term can be (uniquely) decomposed into $\mathtt{D}\langle v\rangle$, where $\mathtt{D}$ is a distant context and $v$ a value. Thus for example, let $t := x_1(u, y.x_2(y, z.z))$. Then, there are three possible decompositions of $t$ in terms of distance contexts: $t = \mathtt{D}_0\langle x_1(u, y.x_2(y, z.z))\rangle$ with $\mathtt{D}_0 = \Diamond$, $t = \mathtt{D}_1\langle x_2(y, z.z)\rangle$ with $\mathtt{D}_1 = x_1(u, y.\Diamond)$ and $t = \mathtt{D}_2\langle z\rangle$ with $\mathtt{D}_2 = x_1(u, y.x_2(y, z.\Diamond))$.

## 2.2 CBN and CBV Operational Semantics

The CBN operational semantics relies on a notion of **right substitution**, which is the expected capture-free meta-level substitution on $\mathtt{T}_J$-terms:.

$$\begin{array}{rlcl} & \{u/x\}x & := & u \\ (x \neq y) & \{u/x\}y & := & y \end{array} \qquad \begin{array}{rcl} \{u/x\}(\lambda y.t) & := & \lambda y.\{u/x\}t \\ \{u/x\}(t(s,y.r)) & := & (\{u/x\}t)(\{u/x\}s, y.\{u/x\}r) \end{array}$$

The CBV operational semantics is based on **left substitution**:

$$\{v\backslash\!\!\backslash x\}t \;:=\; \{v/x\}t \qquad\qquad \{s(u,y.r)\backslash\!\!\backslash x\}t \;:=\; s(u,y.\{r\backslash\!\!\backslash x\}t)$$

Notice however that left substitution of a value invokes the right substitution, and left substitution of a generalized application performs a commutative/permutative conversion. Thus for example, $\{\mathtt{I}\backslash\!\!\backslash x\}(x(\mathtt{I},y.y)) = \{\mathtt{I}/x\}(x(\mathtt{I},y.y)) = \mathtt{I}(\mathtt{I},y.y)$ and $\{\mathtt{I}(\mathtt{I},y.y)\backslash\!\!\backslash z'\}z = \mathtt{I}(\mathtt{I},y.\{y\backslash\!\!\backslash z'\}z) = \mathtt{I}(\mathtt{I},y.\{y/z'\}z) = \mathtt{I}(\mathtt{I},y.z)$. Given the unique decomposition of a term $u$ into $\mathtt{D}\langle v\rangle$, we can alternatively define left substitution as: $\{u\backslash\!\!\backslash x\}t = \{\mathtt{D}\langle v\rangle\backslash\!\!\backslash x\}t := \mathtt{D}\langle\{v/x\}t\rangle$. This alternative definition highlights the principle of CBV, which consists of only substituting values. Thus, in the last example, although $z'$ is different from $z$, the context $\mathtt{I}(\mathtt{I},y.\Diamond)$ is not erased, as it is pushed out of the substitution. Notice that $\{u\backslash\!\!\backslash x\}v$ is a value if and only if $u$ is a value. Notice also that $\{t\backslash\!\!\backslash x\}x = t$.

Before giving the reduction rules of our calculi, we first define some general notations. We denote a **reduction rule** $\mathtt{r} \subseteq \mathtt{T}_J \times \mathtt{T}_J$ as $\mapsto_{\mathtt{r}}$. In this paper, **reduction relations**, denoted by $\to_{\mathcal{R}}$, are generated by a finite set of reduction rules closed under some particular contexts. Given a reduction relation $\to_{\mathcal{R}}$, we write $\to^*_{\mathcal{R}}$ (resp. $\to^+_{\mathcal{R}}$) for the reflexive-transitive (resp. transitive) closure of $\to_{\mathcal{R}}$. A term $t$ is said to be in **$\mathcal{R}$-normal form** (written $\mathcal{R}$-nf) iff there is no $t'$ such that $t \to_{\mathcal{R}} t'$.

The reduction relation $\to_{\mathtt{n}}$ of the original CBN calculus $\Lambda J_n$ [24, 25] is defined as the closure under all contexts of the following two reduction rules $\beta$ and $\pi$. The reduction relation $\to_{\mathtt{v}}$ of the original CBV calculus $\Lambda J_v$ [16] is defined as the closure under all contexts of rules $\beta_{\mathtt{v}}$ and $\pi$.

$$\begin{array}{rrcl} (\textbf{CBN } \boldsymbol{\beta}\textbf{-Rule}) & (\lambda x.t)(u,y.r) & \mapsto_{\beta} & \{\{u/x\}t/y\}r \\ (\textbf{CBV } \boldsymbol{\beta}_{\mathtt{v}}\textbf{-Rule}) & (\lambda x.t)(u,y.r) & \mapsto_{\beta_{\mathtt{v}}} & \{\{u\backslash\!\!\backslash x\}t\backslash\!\!\backslash y\}r \\ (\textbf{Permutative } \boldsymbol{\pi}\textbf{-Rule}) & t(u,x.r)(u',y.r') & \mapsto_{\pi} & t(u,x.r(u',y.r')) \end{array}$$

We use instead a *distant* operational semantics, where permutations are directly integrated in the primary CBN/CBV rule of the calculus. We do not dispense with permutations altogether, as they are necessary to unblock $\beta/\beta_{\mathtt{v}}$-redexes when the function is not exactly next to its argument. This way, the distant paradigm focuses on the *computational* content of the calculus by using demand-driven permutations, only when primary redexes are blocked.

▶ **Definition 1.** *The **CBN calculus** $\lambda J_n$ is defined by the grammar $\mathtt{T}_J$ equipped with the **distant call-by-name** reduction relation $\to_{\mathtt{dn}}$ [18]. The relation $\to_{\mathtt{dn}}$ is generated by the closure under all contexts $\mathtt{C}$ of the reduction rule: $\mathtt{D}\langle\lambda x.t\rangle(u,y.r) \mapsto_{\mathtt{d}\beta} \{\mathtt{D}\langle\{u/x\}t\rangle/y\}r$.*

An intuitive explanation of this rule can be given through the previous informal translation of generalized applications to let-bindings: $\mathtt{D}\langle\lambda x.t\rangle(u,y.r)$ corresponds to $\mathtt{let}\ y = \mathtt{D}\langle\lambda x.t\rangle u\ \mathtt{in}\ r$. In this first term, the computation in the foreground comes from the function $\mathtt{D}\langle\lambda x.t\rangle$ and its argument $u$. We get an intermediate result by substituting $u$ for $x$ in $t$ inside the distant context $\mathtt{D}$, thus obtaining $\mathtt{let}\ y = \mathtt{D}\langle\{u/x\}t\rangle\ \mathtt{in}\ r$. This intermediate result can then be fed

to the continuation by unfolding the let-binding, which means substituting it for $y$ in $r$, thus obtaining the contractum $\{\mathtt{D}\langle\{u/x\}t\rangle/y\}r$. Both the distant context and the term $\{u/x\}t$ may be duplicated, or, on the contrary, simply erased, such as in this example:

$$t_0 = (\lambda x.x(\mathtt{I}, y.y))(\mathtt{I}, z'.z) \rightarrow_{\mathtt{dn}} \{\{\mathtt{I}/x\}(x(\mathtt{I}, y.y))/z'\}z = \{\mathtt{I}(\mathtt{I}, y.y)/z'\}z = z$$

Although the original CBN calculus $\Lambda J_n$ is based on rules $\beta$ and $\pi$, the distant CBN calculus $\lambda J_n$ integrates a different permutative rule $\mathtt{p2}$ to $\beta$, in the sense that $\mathtt{d}\beta$ can be generated by several $\mathtt{p2}$-steps followed by a $\beta$-step, where:

(**Permutative $\mathtt{p2}$-Rule**) $\quad t(u, y.\lambda x.r) \mapsto_{\mathtt{p2}} \lambda x.t(u, y.r)$.

Integrating the $\pi$-rule to define a distant CBN calculus would give instead the alternative rule $\mathtt{D}\langle\lambda x.t\rangle(u, y.r) \mapsto \mathtt{D}\langle\{\{u/x\}t/y\}r\rangle$, where the distant context $\mathtt{D}$ is neither duplicated nor erased: there is exactly one occurrence of $\mathtt{D}$ in the contractum. But this feature corresponds to a CBV behavior, and is not quantitatively well-behaved for a CBN semantics (see [18]). This same remark is true for the original CBN calculus $\Lambda J_n$.

▶ **Definition 2.** *The **CBV calculus** $\lambda J_v$ is defined by the grammar $\mathtt{T}_J$ equipped with the **distant call-by-value** reduction relation $\rightarrow_{\mathtt{dv}}$. The relation $\rightarrow_{\mathtt{dv}}$ is generated by the closure under all contexts $\mathtt{C}$ of the reduction rule: $\mathtt{D}\langle\lambda x.t\rangle(u, y.r) \mapsto_{\mathtt{d}\beta_{\mathtt{v}}} \mathtt{D}\langle\{\{u\backslash\!\!\backslash x\}t\backslash\!\!\backslash y\}r\rangle$.*

Notice that we also use a distance semantics for CBV, whereas the original $\Lambda J_v$-calculus relies on rules $\pi$ and $\beta_{\mathtt{v}}$ ($\mathtt{d}\beta_{\mathtt{v}}$ where $\mathtt{D}$ is empty). Both reductions $\rightarrow_{\mathtt{dn}}$ and $\rightarrow_{\mathtt{dv}}$ enjoy confluence. It is also worth noticing that CBN and CBV reducible expressions (*a.k.a. redexes*) are the same. In particular, no $\mathtt{d}\beta_{\mathtt{v}}$-redex is stuck because the argument is not a value. This is a major difference with most CBV calculi. However, the right-hand sides of the rules $\mathtt{d}\beta$ and $\mathtt{d}\beta_{\mathtt{v}}$ differ in two ways: the kind of substitution used (right for CBN, left for CBV) and the fact that $\mathtt{D}$ might be erased or duplicated in CBN, but not in CBV. To illustrate the first difference, we sketch the CBV reduction of the previous term $t_0$:

$$\begin{aligned} t_0 &= (\lambda x.x(\mathtt{I}, y.y))(\mathtt{I}, z'.z) \\ &\mapsto_{\mathtt{d}\beta_{\mathtt{v}}} \{\{\mathtt{I}\backslash\!\!\backslash x\}(x(\mathtt{I}, y.y))\backslash\!\!\backslash z'\}z = \{\mathtt{I}(\mathtt{I}, y.y)\backslash\!\!\backslash z'\}z = \mathtt{I}(\mathtt{I}, y.z) = (\lambda x.x)(\mathtt{I}, y.z) \\ &\mapsto_{\mathtt{d}\beta_{\mathtt{v}}} \{\{\mathtt{I}\backslash\!\!\backslash x\}x\backslash\!\!\backslash y\}z = \{\mathtt{I}\backslash\!\!\backslash y\}z = z \end{aligned}$$

In line two, the substitutions are exactly the ones already detailed after the definition of left substitution. The two (nested) left substitutions on the last line act on values, so that they fall back to the usual right substitution. While in CBN a single step was sufficient to reach the normal form $z$, in CBV we need one additional step.

As usual, CBV does not duplicate computations (outside abstractions), but tries to reduce every argument to a value, and this may create divergent computations. Take for instance $t_1 = \delta(\delta, z.y)$. In CBN, $t_1$ normalizes to $y$, while in CBV $t_1$ loops indefinitely.

$$\begin{aligned} &(\text{CBN}) \quad \delta(\delta, z.y) \mapsto_{\mathtt{d}\beta} \{\{\delta/x\}(x(x, z.z))/z\}y = \{\delta(\delta, z.z)/z\}y = y \\ &(\text{CBV}) \quad \delta(\delta, z.y) \mapsto_{\mathtt{d}\beta_{\mathtt{v}}} \{\{\delta\backslash\!\!\backslash x\}(x(x, z.z))\backslash\!\!\backslash z\}y \\ &\qquad\qquad = \{\{\delta/x\}(x(x, z.z))\backslash\!\!\backslash z\}y = \{\delta(\delta, z.z)\backslash\!\!\backslash z\}y = \delta(\delta, z.\{z\backslash\!\!\backslash z\}y) = \delta(\delta, z.y) \end{aligned}$$

We illustrate the different uses of distant contexts in CBN and CBV on a last example. Let $t_2 = x_1(u, x_2.\lambda x.x)(v, y.y(y, z.z))$, that can be written as $\mathtt{D}\langle\lambda x.x\rangle(v, y.y(y, z.z))$, where $\mathtt{D} = x_1(u, x_2.\Diamond)$. Using the distant semantics, the term $v$ turns out to be an argument of the function $\lambda x.x$, and thus the redex can be fired, although both terms are separated by the (distant) context $\mathtt{D}$:

$$\begin{array}{llll}
\text{(CBN)} & t_2 \to^*_{\mathtt{d}\beta} & \{\mathtt{D}\langle\{v/x\}x\rangle/y\}(y(y,z.z)) = \{\mathtt{D}\langle v\rangle/y\}(y(y,z.z)) \\
& = & \mathtt{D}\langle v\rangle(\mathtt{D}\langle v\rangle, z.z) = x_1(u,x_2.v)(x_1(u,x_2.v),z.z) \\
\text{(CBV)} & t_2 \to^*_{\mathtt{d}\beta_{\mathtt{v}}} & \mathtt{D}\langle\{\{v\backslash\!\backslash x\}x\backslash\!\backslash y\}(y(y,z.z))\rangle = \mathtt{D}\langle\{v\backslash\!\backslash y\}(y(y,z.z))\rangle \\
& = & \mathtt{D}\langle v(v,z.z)\rangle = x_1(u,x_2.v(v,z.z))
\end{array}$$

The way distant contexts are handled in both variants is reminiscent of how the result of the $\beta$-reduction is treated by the two different substitution operations, as the applications inside the distant context are duplicated or erased only in CBN. This difference is justified by the use of distinct underlying permutation rules: $\mathtt{p2}$ for CBN and $\pi$ for CBV.

As a last remark, notice that $\beta_{\mathtt{v}}$-reduction preserves $\pi$-normal forms. Thus, an alternative to distant CBV reduction is to "pre-process" all terms by using full $\pi$-normalization, so that neither distance nor permutations are needed for the CBV computation, which is then defined only on terms in $\pi$-nf. Yet, instead of computing only with $\pi$-nf, we choose to work in a more general framework with arbitrary terms, by applying only those permutations that are necessary to fire $\beta$-redexes, following the same philosophy used in CBN.

## 3 Solvability for Generalized Applications

In this section we give definitions of solvability for CBN and CBV by using the key notion of *head context*. In contrast to the λ-calculus, the syntax of generalized applications makes the identification of the *head* of a term very subtle. In particular, whereas it is possible to use vectorial meta-notations in the λ-calculus, we must use inductive definitions in this framework. **Head contexts** are given by the following grammar:

$$\mathtt{H} ::= \Diamond \mid \lambda x.\mathtt{H} \mid \mathtt{H}(u, x.\mathtt{H}'\langle\!\langle x\rangle\!\rangle) \mid t(u, x.\mathtt{H})$$

While, in general, there are several possibilities to decompose a term into a head context surrounding a subterm, there is a closely related notion of **head variable**, which deterministically distinguishes a particular variable in each term:

$$\frac{}{\mathrm{hv}(x) = x} \qquad \frac{\mathrm{hv}(t) = x}{\mathrm{hv}(\lambda y.t) = x} \qquad \frac{\mathrm{hv}(r) = y \qquad \mathrm{hv}(t) = x}{\mathrm{hv}(t(u,y.r)) = x} \qquad \frac{\mathrm{hv}(r) = x \qquad x \neq y}{\mathrm{hv}(t(u,y.r)) = x}$$

In the third rule we assume w.l.o.g. that $y$ is not bound in $r$. Notice that the head variable may be either free or bound, since $y$ can be equal to $x$ in the second rule. To understand the last two rules, we use the previous analogy with let-bindings. To an application $t(u, y.r)$ corresponds a binding `let` $y$ `=` $tu$ `in` $r$, and to find the head variable of this term, we look inside $r$. For instance, the head variable of `let` $x$ `=` $zz$ `in` $y$, corresponding to $z(z, x.y)$, is $y$. But if we take `let` $x$ `=` $zz$ `in` $x$, corresponding to $z(z, x.x)$, its head variable is $z$. Thus, the head variable of a term with generalized applications is the head variable of the corresponding term where all the let-binding have been unfolded. In the example, $z$ is the head variable of $z(z, x.x)$ because $x$ is itself the head variable of the subterm $x$ inside the continuation.

Given a term $t$ verifying $\mathrm{hv}(t) = x$, there is a unique head context $\mathtt{H}$ such that (1) $t = \mathtt{H}\langle x\rangle$, or (2) $t = \mathtt{H}\langle\!\langle x\rangle\!\rangle$ if $x \in \mathrm{fv}(t)$. Thus for example, given $t_3 := z(z, x.y)$, we have $\mathrm{hv}(t_3) = y$ as well as $t_3 = \mathtt{H}\langle\!\langle y\rangle\!\rangle$ with $\mathtt{H} = z(z, x.\Diamond)$. Given $t_3' := z(z, x.x)$, we have $\mathrm{hv}(t_3') = z$ as well as $t_3' = \mathtt{H}'\langle\!\langle z\rangle\!\rangle$ with $\mathtt{H}' = \Diamond(z, x.\mathtt{H}_0\langle\!\langle x\rangle\!\rangle)$ and $\mathtt{H}_0 = \Diamond$. An example where the head variable is bound is $\mathrm{hv}(\lambda y.t_3) = y$, where $\lambda y.t_3 = \mathtt{H}''\langle y\rangle$ and $\mathtt{H}'' = \lambda y.z(z, x.\Diamond)$.

▶ **Definition 3** (Solvability). *Let $t \in \mathtt{T}_J$. Then:*
*$t$ is **CBN-solvable** iff there is a head (resp. distant) context* $\mathtt{H}$ *and* $\mathtt{D}$ *such that* $\mathtt{H}\langle t\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{D}\langle\mathtt{I}\rangle$.
*$t$ is **CBV-solvable** iff there is a head context* $\mathtt{H}$ *such that* $\mathtt{H}\langle t\rangle \rightarrow^*_{\mathtt{dv}} \mathtt{I}$.

In the CBN λ-calculus, several equivalent definitions of solvability for a λ-term $M$ coexist. In particular: (1) there is a head context $\mathtt{H}$ such that $\mathtt{H}\langle M\rangle \rightarrow^*_\beta \mathtt{I}$; (2) for all λ-term $N$, there is a head context $\mathtt{H}$ such that $\mathtt{H}\langle M\rangle \rightarrow^*_\beta N$. The head contexts here are the usual head contexts of the λ-calculus. The proof that (1) implies (2) is trivial, since $\mathtt{I}N \rightarrow_\beta N$ always hold in the CBN λ-calculus. On the contrary, in Plotkin's original CBV, the two formulations are not equivalent [20], since $\mathtt{I}N$ only reduces to $N$ when $N$ is a value. In our CBV framework, the equivalence of these alternative definitions is straightforward: for any $\mathtt{T}_J$-term $u$, $\mathtt{I}(u, z.z) \mapsto_{\mathtt{d}\beta_\mathtt{v}} u$ always holds. Moreover, our definition of CBN-solvability for a term $t$ is equivalent to an alternative one stating that for all $\mathtt{T}_J$-term $u$ there exist $\mathtt{H}$ and $\mathtt{D}$ such that $\mathtt{H}\langle t\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{D}\langle u\rangle$. Indeed, $\mathtt{D}\langle\mathtt{I}\rangle(u, z.z) \mapsto_{\mathtt{d}\beta} \mathtt{D}\langle u\rangle$ for any $u$.

Notice in our definition of CBN-solvability that the reduction yields an identity plugged inside a distant context, and not just an identity alone. Take *e.g.* the term $t_4 = \Omega(y, z.\mathtt{I})$ containing a non-relevant continuation, as $z \notin \mathrm{fv}(\mathtt{I})$. In the λ-calculus, $t_4$ translates to $(\lambda z.\mathtt{I})(\Omega y)$, which is solvable since $(\lambda z.\mathtt{I})(\Omega y) \rightarrow_\beta \mathtt{I}$. This suggests introducing a garbage collection-like rule for generalized applications which reduces in this case $\Omega(y, z.\mathtt{I}) \rightarrow_{\mathtt{gc}} \mathtt{I}$. This would be consistent with different models of CBN, such as our quantitative type system. However, we prefer to avoid such ad-hoc solution, which can be simply seen as an implementation detail, as it does not change the operational and denotational behavior of terms.

Now, why does our notion of CBV solvability not use this distant context? Take again the term $t_4 = \Omega(y, z.\mathtt{I})$ and its translated λ-term $(\lambda z.\mathtt{I})(\Omega y)$. CBV reduction in the λ-calculus loops on the argument $\Omega y$, that could only be erased if $\Omega y$ is reduced to a value. Therefore, having a definition of solvability which reduces to $\mathtt{D}\langle\mathtt{I}\rangle$ in CBV would be too liberal, and incoherent with the λ-calculus and its associated models.

Although the two definitions of CBN/CBV solvability are slightly different, they both share the same notion of head context, which is independent from the operational semantics.

*Head reduction* in the λ-calculus is the reduction relation generated by the closure of the rule $\beta$ under *head* contexts. A key (operational) property is that a term $t$ is CBN solvable *iff* $t$ normalizes for the head reduction, that is why we say that it is a *solvable reduction*. In Subsec. 4.1, we define a notion of solvable reduction for $\lambda J_n$, and show that it also has this crucial property. For $\lambda J_v$, we will see in Subsec. 5.1 that the solvable relation is bigger than plain CBN solvable reduction. Nevertheless, the solvable reduction we give mimics the behavior of a solvable reduction for the CBV λ-calculus.

## 4 Call-by-Name Solvability

This section is organized in two parts. We first give an operational characterization of solvability through a solvable reduction relation, and then a quantitative type system providing a logical characterization of it.

### 4.1 Operational Characterization of CBN Solvability

The following solvable reduction relation for $\lambda J_n$ plays the same role as the head reduction plays for the λ-calculus.

▶ **Definition 4.** *The **CBN solvable reduction** $\rightarrow_{\mathtt{sn}}$ is defined as the closure of the following reduction rule under head contexts.*

$$\mathtt{D}\langle \lambda x.t\rangle(u, y.\mathtt{H}\langle\langle y\rangle\rangle) \mapsto_{\mathtt{d}\beta\mathtt{h}} \{\mathtt{D}\langle\{u/x\}t\rangle/y\}\mathtt{H}\langle\langle y\rangle\rangle$$

Notice that the solvable reduction is not based on the full $\mathtt{d}\beta$ rule, as it only reduces redexes to which (one of the hereditary) head variables is bound. Thus, the term $t_4 = \Omega(y, z.\mathtt{I})$ is $\mathtt{sn}$-normal but not $\mathtt{d}\beta$-normal.

▶ **Lemma 5.** *The following grammar* $\mathrm{NF}_{\mathtt{sn}}$ *characterizes* $\mathtt{sn}$*-nfs.*

| | |
|---|---|
| ***(CBN Neutral Normal Contexts)*** | $\mathtt{G} ::= \Diamond \mid \mathtt{G}(u, x.\mathtt{G}\langle\langle x\rangle\rangle) \mid t(u, y.\mathtt{G})$ |
| ***(CBN Solvable Normal Terms)*** | $\mathrm{NF}_{\mathtt{sn}} ::= x \mid \lambda x.\,\mathrm{NF}_{\mathtt{sn}} \mid \mathtt{G}\langle\langle x\rangle\rangle(u, y.\,\mathrm{NF}_{\mathtt{sn}})$ |
| | $\mid t(u, x.\,\mathrm{NF}_{\mathtt{sn}})$ *where* $x \neq \mathrm{hv}(\mathrm{NF}_{\mathtt{sn}})$ |

The previous grammar $\mathrm{NF}_{\mathtt{sn}}$ is used to show the following key result:

▶ **Lemma 6.** *Let $t$ be an* $\mathtt{sn}$*-normalizable term. Then $t$ is CBN solvable.*

To give some intuition about the proof, we know $t$ $\mathtt{sn}$-normalizable implies there is some $\mathtt{sn}$-normal form $t'$ such that $t \rightarrow^*_{\mathtt{sn}} t'$. The goal of the proof is to construct a head context $\mathtt{H}$ such that $\mathtt{H}\langle t'\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{D}\langle\mathtt{I}\rangle$ for some $\mathtt{D}$. This is done by induction on the grammar $\mathrm{NF}_{\mathtt{sn}}$. Therefore $t$ is solvable since $\mathtt{H}\langle t\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{H}\langle t'\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{D}\langle\mathtt{I}\rangle$.

▶ **Example 7.** Let $t_5 = y_1(\mathtt{I}, z_1.x)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3) \in \mathrm{NF}_{\mathtt{sn}}$. Notice that $\mathrm{hv}(t_5) = x$. We take $\mathtt{H} = (\lambda x.\Diamond)(\mathbf{o}^1, z.z)(\mathtt{I}, z.z)$ (remember that $\mathbf{o}^1 = \lambda x_1\lambda x_0.x_0$). Then,

$$\begin{aligned}
\mathtt{H}\langle t_5\rangle &= (\lambda x.y_1(\mathtt{I}, z_1.x)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3))(\mathbf{o}^1, z.z)(\mathtt{I}, z.z)\\
&\rightarrow_{\mathtt{dn}} y_1(\mathtt{I}, z_1.\,\mathbf{o}^1)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3)(\mathtt{I}, z.z)\\
&\rightarrow_{\mathtt{dn}} (\lambda y.y_1(\mathtt{I}, z_1.\,\mathbf{o}^0))(\mathtt{I}, z.z)\\
&\rightarrow_{\mathtt{dn}} y_1(\mathtt{I}, z_1.\,\mathbf{o}^0)
\end{aligned}$$

Taking $\mathtt{D} = y_1(\mathtt{I}, z_1.\Diamond)$, we get a term of the expected form $\mathtt{D}\langle\mathtt{I}\rangle$ (since $\mathbf{o}^0 = \mathtt{I}$).

## 4.2 Logical Characterization of CBN Solvability

We now give a type system, called $\cap N$, in which typability and normalization of solvable reduction coincide, *i.e.* not only does typability imply normalization, but the converse implication also holds. In system $\cap N$, terms can be assigned several types. However, they are not typed with sets – which would give an idempotent intersection type system [13] – , but with multisets, which give a *quantitative* flavor to the type system. Also, non-idempotent types justify the use of rule $\mathtt{p2}$, instead of $\pi$, to obtain a distant notion for CBN with generalized applications which is quantitatively well behaved ([18]). Moreover, as in other calculi with non-idempotent intersection types, the proofs of normalization are notably simplified: reducibility or computability arguments can be replaced by simple combinatorial proofs based on the fact that the size of the type derivations strictly decreases along each solvable $\mathtt{sn}$-step.

Given a countable infinite set $BTV$ of base type variables $a, b, c, \ldots$, we define the following sets of types:

| | | | |
|---|---|---|---|
| **(Types)** | $\sigma, \tau, \rho$ | $::=$ | $a \in BTV \mid \mathcal{M} \rightarrow \sigma$ |
| **(Multiset types)** | $\mathcal{M}, \mathcal{N}$ | $::=$ | $[\sigma_i]_{i\in I}$ where $I$ is a finite set |

The empty multiset is denoted $[\,]$. **Environments**, written $\Gamma, \Delta, \Lambda$, are functions from variables to multiset types assigning the empty multiset to all but a finite set of variables. A **typing** is a pair of an environment and a type. The domain of $\Gamma$ is given by $\mathrm{dom}(\Gamma) := \{x \mid \Gamma(x) \neq [\,]\}$. The **union of environments**, written $\Gamma \wedge \Delta$, is defined by $(\Gamma \wedge \Delta)(x) := \Gamma(x) \sqcup \Delta(x)$, where $\sqcup$ denotes multiset union. This notion is extended to several environments as expected, so that $\wedge_{i \in I} \Gamma_i$ denotes a finite union of environments ($\wedge_{i \in I} \Gamma_i$ is to be understood as the empty environment when $I = \emptyset$). We write $\Gamma \setminus\!\!\setminus x$ for the environment such that $(\Gamma \setminus\!\!\setminus x)(y) = \Gamma(y)$ if $y \neq x$ and $(\Gamma \setminus\!\!\setminus x)(x) = [\,]$. We write $\Gamma; \Delta$ for $\Gamma \wedge \Delta$ when $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. A **sequent** has the form $\Gamma \vdash t : \sigma$, where $(\Gamma, \sigma)$ is a typing and $t$ is a term.

$$\frac{}{x : [\sigma] \vdash x : \sigma}\ (\textsc{var}) \qquad \frac{\Gamma; x : \mathcal{M} \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \mathcal{M} \to \sigma}\ (\textsc{abs}) \qquad \frac{(\Gamma_i \vdash t : \sigma_i)_{i \in I}}{\wedge_{i \in I} \Gamma_i \vdash t : [\sigma_i]_{i \in I}}\ (\textsc{many})$$

$$\frac{\Gamma \vdash t : [\mathcal{M}_i \to \sigma_i]_{i \in I} \qquad \Delta \vdash u : \sqcup_{i \in I} \mathcal{M}_i \qquad \Lambda; x : [\sigma_i]_{i \in I} \vdash r : \tau}{\Gamma \wedge \Delta \wedge \Lambda \vdash t(u, x.r) : \tau}\ (\textsc{app})$$

**Figure 1** System $\cap N$.

The quantitative type system $\cap N$ is defined in Fig. 1. Notice that the system is relevant, as there is no weakening. It is a natural extension of Gardner's [21] and De Carvalho's [14] systems to generalized applications. Rule (MANY) may give the empty multiset to any term (case $I = \emptyset$), so being typable with $[\,]$ means in fact being untyped. The interesting rule is (APP), where both $t$ and $u$ are assigned multiset types, since $x$ is not necessarily linear in $r$. Because $u$ is the argument of $t$, it is assigned all the types on the left of the arrow of $t$. For $n \in \mathbb{N}$, we write $\Gamma \Vdash^n_{\cap N} t : \sigma$ or simply $\Gamma \Vdash^n t : \sigma$ if there is a derivation in system $\cap N$ ending in $\Gamma \vdash t : \sigma$ and containing $n$ occurrences of rule (APP). This derivation measure is sufficient to capture the fact that each sn-step deletes at least one (APP) rule (*c.f.* Lem. 10).

▶ **Example 8.** Take again $t_4 = \Omega(y, z.\mathtt{I})$ (we expand $\mathtt{I}$ to $\lambda x.x$ in the derivation). Although the evaluation of the subterm $\Omega$ is not terminating (and thus $\Omega$ can only be typed with the empty multiset), $t_4$ is typable:

$$\frac{\dfrac{}{\vdash \Omega : [\,]}\ (\textsc{many}) \qquad \dfrac{}{\vdash y : [\,]}\ (\textsc{many}) \qquad \dfrac{\dfrac{}{x : [\sigma] \vdash x : \sigma}\ (\textsc{var})}{\vdash \lambda x.x : [\sigma] \to \sigma}\ (\textsc{abs})}{\vdash \Omega(y, z.\lambda x.x) : [\sigma] \to \sigma}\ (\textsc{app})$$

We now prove that terms typable in $\cap N$ are exactly the ones that normalize with $\to_{\mathtt{sn}}$. The proof relies on two key lemmas.

1. *Weighted subject reduction* does not only state that the typing of a term is preserved along reduction – as usual – , but also that the size of the typing derivation decreases at each sn-step. From this we can deduce that typable terms normalize with $\to_{\mathtt{sn}}$.
2. *Subject expansion* is the opposite of subject reduction: if $t$ reduces to a term $t'$ and $t'$ is typable, then $t$ is also typable with the same typing as $t'$. It is also possible to attach quantitative information to the subject expansion lemma, proving that the size of the type derivation of $t'$ is smaller than the one of $t$, but this will not be useful here.

Subject reduction and subject expansion give soundness and completeness of $\to_{\mathtt{dn}}$ w.r.t. to the type system. This is a peculiarity of intersection type systems, which makes it possible to derive a denotational semantics from the typing itself. In the case of quantitative type systems, it is a relational model, *i.e.* a model in the category **Rel** of sets and relations [9].

## Soundness

To prove subject reduction, we first need to prove a *substitution lemma*, as usual. Because we are in a quantitative model, this lemma also relates the sizes of the corresponding derivations.

▶ **Lemma 9** (Substitution for $\cap N$). *If* $\Gamma; x : \mathcal{M} \Vdash^{n}_{\cap N} t : \sigma$ *and* $\Delta \Vdash^{m}_{\cap N} u : \mathcal{M}$*, then there is a derivation* $\Gamma \wedge \Delta \Vdash^{m+n}_{\cap N} \{u/x\}t : \sigma$*.*

▶ **Lemma 10** (Weighted Subject Reduction for $\cap N$). *If* $\Gamma \Vdash^{n_1}_{\cap N} t_1 : \sigma$ *and* $t_1 \to_{\mathtt{sn}} t_2$*, then* $\Gamma \Vdash^{n_2}_{\cap N} t_2 : \sigma$ *with* $n_1 > n_2$*.*

**Proof.** The proof is by induction on the reduction step $t_1 \to_{\mathtt{sn}} t_2$. The base case is $t_1 = \mathtt{D}\langle \lambda x.t\rangle(u, y.\mathtt{H}\langle\langle y\rangle\rangle) \mapsto_{\mathtt{sn}} \{\mathtt{D}\langle\{u/x\}t\rangle/y\}\mathtt{H}\langle\langle y\rangle\rangle = t_2$, which we decompose into a series of $\mathtt{p2}$-permutation steps followed by a $\beta\mathtt{h}$-step (a $\mathtt{d}\beta\mathtt{h}$-step with an empty distant context): $t_1 \mapsto^{*}_{\mathtt{p2}} (\lambda x.\mathtt{D}\langle t\rangle)(u, y.\mathtt{H}\langle\langle y\rangle\rangle) \mapsto_{\beta\mathtt{h}} t_2$. We prove subject reduction independently for $\mathtt{p2}$ and $\beta\mathtt{h}$. For the last one, we apply the substitution lemma twice. The permutative and the inductive cases (reduction under head contexts) are straightforward by the induction hypothesis. ◀

The size of type derivations is a natural number decreasing at every step, so that soundness is a direct corollary. Notice that no reducibility proof is needed.

▶ **Corollary 11** (Soundness for $\lambda J_n$). *If* $\Gamma \Vdash^{n}_{\cap N} t : \sigma$*, then* $t$ *is* $\mathtt{sn}$*-normalizable and the number of* $\mathtt{sn}$*-steps needed to normalize* $t$ *is bounded by* $n$*.*

## Completeness

To show that every $\mathtt{sn}$-normalizable term is typable, we need subject expansion. This property, like subject reduction, needs a preliminary lemma, this time *anti-substitution*.

▶ **Lemma 12** (Anti-Substitution for $\cap N$). *If* $\Gamma \Vdash \{u/x\}t : \sigma$*, then there exists* $\Gamma_t$*,* $\Gamma_u$ *and* $\mathcal{M}$ *such that* $\Gamma_t; x : \mathcal{M} \Vdash t : \sigma$*,* $\Gamma_u \Vdash u : \mathcal{M}$ *and* $\Gamma = \Gamma_t \wedge \Gamma_u$*.*

▶ **Lemma 13** (Subject Expansion for $\cap N$). *If* $\Gamma \Vdash_{\cap N} t_2 : \sigma$ *and* $t_1 \to_{\mathtt{dn}} t_2$*, then* $\Gamma \Vdash_{\cap N} t_1 : \sigma$*.*

**Proof.** Notice that the statement is about full $\mathtt{dn}$ reduction, which is useful in the proof of Thm. 16. The proof is by induction on $t_1 \to_{\mathtt{dn}} t_2$ and uses the anti-substitution lemma. ◀

Another component of the completeness proof is the fact that $\mathtt{sn}$-normal forms are typable.

▶ **Lemma 14** (Typing $\mathtt{sn}$-nfs). *Let* $t \in \mathrm{NF}_{\mathtt{sn}}$*. Then there exists* $\sigma$ *such that*
1. *If* $t = \mathtt{H}\langle\langle x\rangle\rangle$ *for some* $x$*, then there is* $\tau$ *such that* $x : [\tau] \Vdash t : \sigma$*.*
2. *Otherwise,* $\emptyset \Vdash t : \sigma$*.*

▶ **Corollary 15** (Completeness for $\lambda J_n$). *Let* $t \in \mathtt{T}_J$ *be* $\mathtt{sn}$*-normalizable. Then* $t$ *is typable in system* $\cap N$*.*

**Proof.** By definition, the term $t$ is reducible to a $\mathtt{sn}$-normal form $t'$. By Lem. 14, $t'$ is typable. Subject Expansion gives typability of $t$. ◀

### Characterization of CBN Solvability

We can now derive the main theorem of this section.

▶ **Theorem 16** (CBN Characterization). *Let $t \in \mathtt{T}_J$. Then $t$ is CBN solvable iff $t$ is $\cap N$-typable iff $t$ is* sn*-normalizable.*

**Proof.** Normalizable $\implies$ solvable holds by Lem. 6. Typable $\implies$ normalizable holds by Cor. 11. For solvable $\implies$ typable: take $t$ solvable, so that there are contexts $\mathtt{H}, \mathtt{D}$ such that $\mathtt{H}\langle t\rangle \rightarrow^*_{\mathtt{dn}} \mathtt{D}\langle \mathtt{I}\rangle$. Since $\mathtt{D}\langle \mathtt{I}\rangle$ is $\cap N$-typable by Lem. 14, and the system $\cap N$ satisfies subject expansion (Lem. 13), then $\mathtt{H}\langle t\rangle$ is $\cap N$-typable, which implies $t$ is $\cap N$-typable. ◀

## 5 Call-by-Value Solvability

As in the previous section, we first give an operational characterization of solvability and then a quantitative type system characterizing it.

### 5.1 Operational Characterization of CBV Solvability

In CBN, the method to get the identity from a term plugged into a head context is to successively erase all the arguments, by replacing the head variable by a projection term $\mathbf{o}^n = \lambda x_n \ldots x_0.x_0$. But in CBV, arguments which are not values cannot be erased: in order to be erased they need to be *potentially valuable.*

▶ **Definition 17.** *A term $t$ is **potentially valuable** iff there exist a distant context $\mathtt{D}$ and a value $v$ such that $\mathtt{D}\langle t\rangle \rightarrow^*_{\mathtt{dv}} v$.*

The distant context in the previous definition can be seen as a list of substitutions – eventually affecting the free variables of $t$ – used to transform $t$ into a value.

▶ **Example 18.** Let $t_6 = x(\Omega, z.z)$. In CBN, it is sufficient to take the head context $\mathtt{H} = \mathtt{I}(\mathbf{o}^1, x.\Diamond)$ so that $\mathtt{H}\langle t_6\rangle \rightarrow_{\mathtt{dn}} \mathbf{o}^1(\Omega, z.z) \rightarrow_{\mathtt{dn}} \mathtt{I}$ simply erases the diverging term $\Omega$. In CBV, however, this is not possible since $\mathtt{H}\langle t_6\rangle \rightarrow_{\mathtt{dv}} \mathbf{o}^1(\Omega, z.z) \rightarrow_{\mathtt{dv}} \delta(\delta, x.\mathtt{I})$, which diverges. The term $t_6$ is only solvable in CBN. On the contrary, the term $x(\lambda y.\Omega, z.z)$ is solvable in both CBN and CBV because the argument $\lambda y.\Omega$ can be erased.

Interestingly, there is a (non-deterministic) reduction relation $\rightarrow_{\mathtt{pv}}$ such that the normalizing terms for $\rightarrow_{\mathtt{pv}}$ are exactly the potentially valuable terms (*c.f.* Thm. 34). It is in fact a *weak* reduction relation in which reduction can occur anywhere but below abstractions. We detail this result before tackling the CBV solvable reduction.

▶ **Definition 19.** *The **valuable reduction relation** $\rightarrow_{\mathtt{pv}}$ is defined by the following rules:*

$$\frac{t \mapsto_{\mathtt{d}\beta_{\mathtt{v}}} t'}{t \rightarrow_{\mathtt{pv}} t'} \qquad \frac{t \rightarrow_{\mathtt{pv}} t'}{t(u, y.r) \rightarrow_{\mathtt{pv}} t'(u, y.r)} \qquad \frac{u \rightarrow_{\mathtt{pv}} u'}{t(u, y.r) \rightarrow_{\mathtt{pv}} t(u', y.r)} \qquad \frac{r \rightarrow_{\mathtt{pv}} r'}{t(u, y.r) \rightarrow_{\mathtt{pv}} t(u, y.r')}$$

▶ **Lemma 20.** *Consider the following grammar:*

$$\begin{array}{llcl} \textbf{\textit{(Valuable Neutral Normal Terms)}} & \mathrm{NE}_{\mathtt{pv}} & ::= & x \mid \mathrm{NE}_{\mathtt{pv}}(\mathrm{NF}_{\mathtt{pv}}, y.\,\mathrm{NE}_{\mathtt{pv}}) \\ \textbf{\textit{(Valuable Normal Terms)}} & \mathrm{NF}_{\mathtt{pv}} & ::= & x \mid \mathrm{NE}_{\mathtt{pv}}(\mathrm{NF}_{\mathtt{pv}}, y.\,\mathrm{NF}_{\mathtt{pv}}) \mid \lambda x.t \end{array}$$

*Then, $t \in \mathrm{NF}_{\mathtt{pv}}$ iff $t$ is in* pv*-normal form.*

This grammar is used to show the following property, whose converse is obtained in Thm. 34.

▶ **Lemma 21.** *Let $t$ be a* pv*-normalizable term. Then $t$ is potentially valuable.*

Note that being potentially valuable is weaker than being solvable, because values are always potentially valuable but not necessarily solvable, like $\lambda x.\Omega$. We are now ready to build the solvable reduction on top of the valuable one.

▶ **Definition 22.** *The **CBV solvable reduction relation** $\to_{\mathtt{sv}}$ is defined as follows:*

$$\frac{t \mapsto_{\mathtt{d}\beta_{\mathtt{v}}} t'}{t \to_{\mathtt{sv}} t'} \qquad \frac{t \to_{\mathtt{sv}} t'}{\lambda x.t \to_{\mathtt{sv}} \lambda x.t'}$$

$$\frac{t \to_{\mathtt{pv}} t'}{t(u, x.r) \to_{\mathtt{sv}} t'(u, x.r)} \qquad \frac{u \to_{\mathtt{pv}} u'}{t(u, x.r) \to_{\mathtt{sv}} t(u', x.r)} \qquad \frac{r \to_{\mathtt{sv}} r'}{t(u, x.r) \to_{\mathtt{sv}} t(u, x.r')}$$

An equivalent formulation can be given by the closure of $\mathtt{d}\beta_{\mathtt{v}}$ under head contexts, plus the first and second rules for closure under application. With these rules, we make sure that in an application $t(u, x.r)$, the subterms $t$ and $u$ are pv-normalizable, and thus potentially valuable. In case there is a divergent term in $u$ or $t$, the solvable reduction will diverge. This relation is strictly bigger than the CBN solvable relation $\to_{\mathtt{sn}}$, as it diverges on more terms.

▶ **Lemma 23.** *Let us consider the following grammar:*

$$\textit{(CBV Solvable Normal Terms)} \qquad \mathrm{NF}_{\mathtt{sv}} ::= x \mid \lambda x.\, \mathrm{NF}_{\mathtt{sv}} \mid \mathrm{NE}_{\mathtt{pv}}(\mathrm{NF}_{\mathtt{pv}}, y.\, \mathrm{NF}_{\mathtt{sv}})$$

*Then, $t \in \mathrm{NF}_{\mathtt{sv}}$ iff $t$ is in* sv*-normal form. Notice that $\mathrm{NF}_{\mathtt{sv}} \subset \mathrm{NF}_{\mathtt{pv}}$.*

▶ **Lemma 24.** *Let $t$ be an* sv*-normalizable term. Then $t$ is CBV solvable.*

A hint of the proof: since $t$ is sv-normalizable, there is some sv-normal form $t$ such that $t \to^*_{\mathtt{sv}} t'$. We must construct a head context $\mathtt{H}$ such that $\mathtt{H}\langle t' \rangle \to^*_{\mathtt{dv}} \mathtt{I}$. This is done by induction on the grammar $\mathrm{NF}_{\mathtt{sv}}$. Therefore, $t$ is solvable since $\mathtt{H}\langle t \rangle \to^*_{\mathtt{dv}} \mathtt{H}\langle t' \rangle \to^*_{\mathtt{dv}} \mathtt{I}$. A difference with Lem. 6 for CBN is that the head context must erase all applications of $t'$, even the ones which are non-relevant, making the proof more involved.

▶ **Example 25.** Once again, the goal is to construct a head context for the sv-normal form of $t$. Take for instance the term $t_5 = y_1(\mathtt{I}, z_1.x)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3)$ from Ex. 7, also in $\mathrm{NF}_{\mathtt{sv}}$. This time, we take $\mathtt{H} = (\lambda y_1.\lambda x.\lambda y_2.\Diamond)(\mathbf{o}^1, z.z)(\mathbf{o}^1, z.z)(\mathbf{o}^1, z.z)(\mathtt{I}, z.z)$. Indeed,

$$\begin{aligned}\mathtt{H}\langle t_5 \rangle &= (\lambda y_1.\lambda x.\lambda y_2.y_1(\mathtt{I}, z_1.x)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3))(\mathbf{o}^1, z.z)(\mathbf{o}^1, z.z)(\mathbf{o}^1, z.z)(\mathtt{I}, z.z)\\ &\to^3_{\mathtt{dv}} \mathbf{o}^1(\mathtt{I}, z_1.\,\mathbf{o}^1)(\mathbf{o}^1(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3)(\mathtt{I}, z.z) \to^3_{\mathtt{dv}} \mathbf{o}^1(\mathbf{o}^1(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3)(\mathtt{I}, z.z)\\ &\to_{\mathtt{dv}} \mathbf{o}^1(\mathtt{I}, z_3.\lambda y.z_3)(\mathtt{I}, z.z) \to_{\mathtt{dv}} (\lambda y.\,\mathbf{o}^0)(\mathtt{I}, z.z) \to_{\mathtt{dv}} \mathbf{o}^0 = \mathtt{I}\end{aligned}$$

## 5.2 Logical Characterization of CBV Solvability

We will now define a quantitative type system characterizing CBV solvability. The grammar of types is different from Subsec. 4.2, as multiset types are considered as types and in particular may also occur on the right hand-side of an arrow.

| | | | |
|---|---|---|---|
| **(Types)** | $\sigma, \tau$ | $::=$ | $a \in BTV \mid \mathcal{M} \mid \mathcal{M} \to \sigma$ |
| **(Multiset types)** | $\mathcal{M}, \mathcal{N}$ | $::=$ | $[\sigma_i]_{i \in I}$ where $I$ is a finite set |

$$\frac{}{x : \mathcal{M} \vdash x : \mathcal{M}} \text{(VAR)} \qquad \frac{(\Gamma_i ; x : \mathcal{M}_i \vdash t : \sigma_i)_{i \in I}}{\wedge_{i \in I} \Gamma_i \vdash \lambda x.t : [\mathcal{M}_i \to \sigma_i]_{i \in I}} \text{(ABS)}$$

$$\frac{\Gamma \vdash t : [\mathcal{M} \to \mathcal{N}] \qquad \Delta \vdash u : \mathcal{M} \qquad \Lambda ; x : \mathcal{N} \vdash r : \sigma}{\Gamma \wedge \Delta \wedge \Lambda \vdash t(u, x.r) : \sigma} \text{(APP)}$$

**Figure 2** System $\cap V$.

We use a unique type system $\cap V$, defined in Fig. 2, to characterize both potential valuability and solvability. The type system is inspired from [10]. Again, we write $\Gamma \Vdash^{n}_{\cap V} t : \sigma$ if the sequent $\Gamma \vdash t : \sigma$ is derivable in this system with a derivation of size $n$ (containing $n$ occurrences of (APP)). We will show that typability in $\cap V$ is equivalent to normalizability of the valuable reduction. To logically characterize solvable terms, we constraint typability to a particular set of types, where the empty multiset type cannot appear anymore on the right-hand sides of arrows. We take this idea from [5], where these types are called *solvable*.

▶ **Definition 26** (Solvable types). *A* ***solvable type*** *is not an empty multiset, and has no empty multiset on the right of an arrow. Formally,*

$$\begin{array}{lrcl} (\textbf{\textit{Solvable types}}) & \sigma^{\mathrm{s}}, \tau^{\mathrm{s}} & ::= & a \in BTV \mid \mathcal{M}^{\mathrm{s}} \mid \mathcal{M} \to \sigma^{\mathrm{s}} \\ (\textbf{\textit{Solvable multiset types}}) & \mathcal{M}^{\mathrm{s}}, \mathcal{N}^{\mathrm{s}} & ::= & [\sigma^{\mathrm{s}}_i]_{i \in I} \quad \textit{where } I \textit{ is a non-empty finite set} \end{array}$$

Unlike CBN, where the empty multiset $[\,]$ is used to mark *untyped* subterms, being typable in CBV with $[\,]$ is equivalent to being potentially valuable. The unsolvable term $\lambda x.\Omega$, for instance, can be typed with $[\,]$ by rule (ABS) with $I$ empty. But it cannot be typed with any other type, and in particular with a solvable one, otherwise $\Omega$ would need to be typable. Notice also that the terms $t$ and $u$ in rule (APP) must always be typed, at least with type $[\,]$. That is why the term $t_4$ of Ex. 8, typable in $\cap N$, is not typable in $\cap V$.

We now prove that terms typable in $\cap V$ are exactly those that are normalizable for the valuable reduction, and among them, those typable with a solvable type are the ones normalizing for the solvable reduction. The proof method is the same as for CBN (Subsec. 4.2), but the statements cover both reduction relations at the same time.

## Soundness

Soundness follows the same scheme used for CBN (no reducibility proof is needed): a weighted subject reduction property, based on a substitution lemma, is used to show that typability implies normalization.

▶ **Lemma 27** (Substitution for $\cap V$). *Let* $\Gamma ; x : \mathcal{M} \Vdash^{n}_{\cap V} t : \sigma$ *and* $\Delta \Vdash^{m}_{\cap V} u : \mathcal{M}$.

- *If $u$ is a value, then* $\Gamma \wedge \Delta \Vdash^{n+m}_{\cap V} \{u/x\}t : \sigma$.
- *For any $u$,* $\Gamma \wedge \Delta \Vdash^{n+m}_{\cap V} \{u\backslash\!\!\backslash x\}t : \sigma$.

▶ **Lemma 28** (Weighted Subject Reduction for $\cap V$). *Let* $\Gamma \Vdash^{n_1}_{\cap V} t_1 : \sigma$ *and* $t_1 \to_{\mathtt{dv}} t_2$. *Then* $\Gamma \Vdash^{n_2}_{\cap V} t_2 : \sigma$ *with* $n_1 \geq n_2$. *Moreover:*

1. *If* $t_1 \to_{\mathtt{pv}} t_2$, *then* $n_1 > n_2$.
2. *If* $t_1 \to_{\mathtt{sv}} t_2$ *and $\sigma$ is a solvable type, then* $n_1 > n_2$.

▶ **Corollary 29** (Soundness for $\cap V$). *Let $\Gamma \Vdash^n_{\cap V} t : \sigma$. Then,*
1. *The term $t$ is* pv*-normalizing and the number of* pv*-steps needed to normalize $t$ is bound by $n$.*
2. *If $\sigma$ is a solvable type, then $t$ is* sv*-normalizing and the number of* sv*-steps needed to normalize $t$ is bound by $n$.*

## Completeness

Completeness also follows the same scheme used for CBN: we show that normal forms are typable, together with a subject expansion property, based on an anti-substitution lemma.

▶ **Lemma 30** (Anti-substitution for $\cap V$).
- *If $\Gamma \Vdash_{\cap V} \{v/x\}t : \sigma$, then there are $\Gamma_t$, $\Gamma_v$ and $\mathcal{M}$ such that $\Gamma_t; x : \mathcal{M} \Vdash_{\cap V} t : \sigma$, $\Gamma_v \Vdash_{\cap V} v : \mathcal{M}$ and $\Gamma = \Gamma_t \wedge \Gamma_v$.*
- *If $\Gamma \Vdash_{\cap V} \{u \backslash\backslash x\}t : \sigma$, then there are $\Gamma_t$, $\Gamma_u$ and $\mathcal{M}$ such that $\Gamma_t; x : \mathcal{M} \Vdash_{\cap V} t : \sigma$, $\Gamma_u \Vdash_{\cap V} u : \mathcal{M}$ and $\Gamma = \Gamma_t \wedge \Gamma_u$.*

▶ **Lemma 31** (Subject Expansion for $\cap V$). *Let $\Gamma \Vdash_{\cap V} t_2 : \sigma$ and $t_1 \to_{\mathtt{dv}} t_2$. Then $\Gamma \Vdash_{\cap V} t_1 : \sigma$.*

▶ **Lemma 32** (Typing $\mathrm{NF}_{\mathtt{pv}}$-nfs). *Let $t \in \mathtt{T}_J$.*
1. *If $t \in \mathrm{NF}_{\mathtt{pv}}$, then there exists $\Gamma$ such that $\Gamma \Vdash_{\cap V} t : [\,]$.*
2. *If $t \in \mathrm{NF}_{\mathtt{sv}}$, then there exist $\Gamma$ and $\sigma^{\mathrm{s}}$ solvable such that $\Gamma \Vdash_{\cap V} t : \sigma^{\mathrm{s}}$.*

▶ **Corollary 33** (Completeness for $\cap V$). *Let $t \in \mathtt{T}_J$.*
1. *If $t$ is* pv*-normalizing, then $t$ is typable in $\cap V$.*
2. *If $t$ is* sv*-normalizing, then $t$ is typable in $\cap V$ with a solvable type.*

## Characterization of CBV Solvability

We can now derive the main theorem of this section.

▶ **Theorem 34** (Characterization). *Let $t \in \mathtt{T}_J$. Then,*
- *$t$ is potentially valuable iff $t$ is $\cap V$-typable iff $t$ is* pv*-normalizable, and*
- *$t$ is CBV solvable iff $t$ is $\cap V$-typable with a solvable type iff $t$ is* sv*-normalizable.*

**Proof.** pv/sv-normalizable $\implies$ potentially valuable/CBV solvable holds respectively by Lem. 21/Lem. 24. Typable/Typable with a solvable type $\implies$ pv/sv-normalizable: both hold by Cor. 29. For potentially valuable/CBV solvable $\implies$ typable: similar to Thm. 16, with the corresponding Lem. 31 and Lem. 32, and using the following two facts: (1) every value is typable and (2) I is typable with a solvable type. ◀

# 6 Extension to $\Lambda J_n$, $\Lambda J_v$ and the $\lambda$-calculus

We have argued in favor of endowing generalized applications with a distant operational semantics: permutations are only used when they are necessary to unblock redexes, thus putting the focus on the computational content on the calculus, and also bringing the operational semantics of the calculus closer to the quantitative model. Nonetheless, this choice should not have an influence on overall properties such as strong normalization, solvability or potential valuability. We also wish to be conservative with respect to the original CBN and CBV calculi $\Lambda J_n$ and $\Lambda J_v$. In this section we show that. More precisely,

we prove the equivalence of CBN/CBV solvability with and without distance using the quantitative type systems introduced in previous sections. We also show that our CBN/CBV notion of solvability is equivalent to the original one for the λ-calculus, a result which is expected but not evident.

## 6.1 Solvability for $\Lambda J_n$ and $\Lambda J_v$

Remember that $\to_{\mathtt{n}}$ (resp. $\to_{\mathtt{v}}$) is the reduction relation associated to the original CBN (resp. CBV) calculus. In what follows we write *local* to mean *non-distant*.

▶ **Definition 35** (Local Solvability). *Let $t \in \mathtt{T}_J$.*

***($\Lambda J_n$)*** *$t$ is **CBN local solvable** iff there is a head context $\mathtt{H}$ and a distant context $\mathtt{D}$ such that $\mathtt{H}\langle t\rangle \to_{\mathtt{n}}^* \mathtt{D}\langle \mathtt{I}\rangle$.*

***($\Lambda J_v$)*** *$t$ is **CBV local solvable** iff there is a head context $\mathtt{H}$ such that $\mathtt{H}\langle t\rangle \to_{\mathtt{v}}^* \mathtt{I}$.*

Notice that the terms $t_4 = \Omega(y, z.\mathtt{I})$ and $t_6 = x(\Omega, z.\mathtt{I})$ are CBN but not CBV locally solvable. The term $t_5 = y_1(\mathtt{I}, z_1.x)(y_2(\mathtt{I}, z_2.z_2), z_3.\lambda y.z_3)$ is both CBN and CBV solvable.

▶ **Definition 36.** *The **CBN local solvable reduction** $\to_{\mathtt{lsn}}$ is generated by the closure of the following rules $\beta\mathtt{h}$ and $\pi\mathtt{h}$ under head contexts.*

$$(\lambda x.t)(u, y.\mathtt{H}\langle\langle y\rangle\rangle) \mapsto_{\beta\mathtt{h}} \{\{u/x\}t/y\}\mathtt{H}\langle\langle y\rangle\rangle$$
$$t(u, x.r)(u', y.\mathtt{H}\langle\langle y\rangle\rangle) \mapsto_{\pi\mathtt{h}} t(u, x.r(u', y.\mathtt{H}\langle\langle y\rangle\rangle))$$

*The **local valuable reduction** $\to_{\mathtt{lpv}}$ and **CBV local solvable reduction** $\to_{\mathtt{lsv}}$ are defined by the closure of rules $\beta_{\mathtt{v}}$ and $\pi$ under the same contexts used in their distant counterparts (Def. 19 and Def. 22 respectively).*

▶ **Theorem 37** (Local Characterization). *Let $t \in \mathtt{T}_J$. Then,*

***CBN:*** *$t$ is CBN local solvable iff $t$ is $\cap N$-typable iff $t$ is $\mathtt{lsn}$-normalizable.*

***CBV:*** *$t$ is CBV local potentially valuable iff $t$ is $\cap V$-typable iff $t$ is $\mathtt{lpv}$-normalizable, and $t$ is CBV local solvable iff $t$ is $\cap V$-typable with a solvable type iff $t$ is $\mathtt{lsv}$-normalizable.*

Since the same notion of typability is used in the distant and local characterizations, this gives the following equivalence for free.

▶ **Corollary 38.** *CBN (resp. CBV) solvability is equivalent to CBN (resp. CBV) local solvability.*

## 6.2 Equivalence with Solvability in the λ-Calculus

We also relate solvability of generalized applications to solvability in the λ-calculus. More precisely, we consider λ-calculi with explicit substitutions. The set of terms with explicit substitutions $\mathtt{T}_{ES}$ is generated by the following grammar:

$$M, N ::= x \mid \lambda x.M \mid MN \mid [N/x]M$$

The last clause is an alternative notation for a let-binding $\mathtt{let}\ x\ \mathtt{=}\ N\ \mathtt{in}\ M$.

We show that the following standard translations preserve solvability in both directions.

▶ **Definition 39** (Translations).

$$(\mathtt{T}_J \mapsto \mathtt{T}_{ES}) \quad x^* := x \quad (\lambda x.t)^* := \lambda x.t^* \quad t(u, x.r)^* := [t^*u^*/x]r^*$$
$$(\mathtt{T}_{ES} \mapsto \mathtt{T}_J) \quad x^\circ := x \quad (\lambda x.M)^\circ := \lambda x.M^\circ$$
$$(MN)^\circ := M^\circ(N^\circ, z.z) \quad ([N/x]M)^\circ := \mathtt{I}(N^\circ, x.M^\circ)$$

Our proofs use quantitative types. For CBN, we call $\mathcal{N}$ the type system in [28, 10], which characterizes head normalization and thus solvability. For CBV, we call $\mathcal{V}$ an alternative but faithful presentation of the type system in [5] for which all and only solvable terms are typable with a solvable type.

Indeed, we show that the translation of a $\mathtt{T}_J$-term typable in system $\cap N$ (resp. $\cap V$) is also typable in system $\mathcal{N}$ (resp. $\mathcal{V}$), and vice-versa.

▶ **Theorem 40** (Preservation of Typing). *Let $t \in \mathtt{T}_J$ and $M \in \mathtt{T}_{ES}$.*

- $\Gamma \Vdash_{\cap N} t : \sigma$ *implies* $\Gamma \Vdash_{\mathcal{N}} t^* : \sigma$ *and* $\Gamma \Vdash_{\mathcal{N}} M : \sigma$ *implies* $\Gamma \Vdash_{\cap N} M^\circ : \sigma$.
- $\Gamma \Vdash_{\cap V} t : \sigma$ *implies* $\Gamma \Vdash_{\mathcal{V}} t^* : \sigma$ *and* $\Gamma \Vdash_{\mathcal{V}} M : \sigma$ *implies* $\Gamma \Vdash_{\cap V} M^\circ : \sigma$.

As CBN/CBV solvability in the λ-calculus is equivalent to $\mathcal{N}$-typability/$\mathcal{V}$-typability with a solvable type, we get the final results:

▶ **Corollary 41.** *Let $t$ be a $\mathtt{T}_J$-term.*

- *$t$ is CBN solvable if and only if $t^*$ is CBN solvable in the λ-calculus.*
- *$t$ is CBV solvable if and only if $t^*$ is CBV solvable in the λ-calculus.*

## 7 Related Works and Conclusion

Solvability for Plotkin's CBV calculus has been first studied in [32], where some key concepts (notably potentially valuable terms and solvable types) are introduced. Another contribution in this same framework is [20], where a genericity lemma is proved. An operational characterization of CBV solvability in terms of CBV reduction appears in [7], for a distant CBV calculus with explicit substitutions. In [12], a similar result is obtained for Plotkin's CBV λ-calculus combined with permutative rules used to unblock redexes. The authors of *op.cit.* also give a relational model of solvability, based on quantitative types. We draw inspiration from all these works to address the challenge of solvability in generalized applications, where stuck redexes are produced by the particular applicative structure of the calculus.

More generally, finding good operational formalisms for call-by-value is an active topic of research (see [4]), with new insights from linear logic [2, 22] and the sequent calculus [23]. We believe that $\lambda J_v$ holds a singular place among them, thanks to its natural way to deal with stuck redexes. An open problem is to find a fully abstract model for the CBV λ-calculus. We would like to see whether generalized applications help in this quest. In particular, it would be interesting to understand CBV approximation for generalized applications, CBV Böhm trees [8, 26] based on the solvable reduction, as well as separability [31].

Idempotent intersection types for $\Lambda J_n$ are proposed in [30]. Although intersection types achieve to characterize all and only strongly normalizable terms, they do not reject $\pi$ as a permutation rule for CBN, albeit quantitatively unsound. Quantitative types for CBN and $\lambda J_n$ are introduced in [18], where only strong normalization has been addressed. Our quantitative CBN and CBV type systems can be seen as the first relational models for generalized applications. They are adapted from [10], but we take the key insight about solvable types from [5]. It would be interesting to see if the techniques developed for *tightness* [3, 29] can also be adapted to this framework.

## References


1 Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Beniamino Accattoli. Proof nets and the call-by-value λ-calculus. *Theoretical Computer Science*, 606:2–24, November 2015. doi:10.1016/j.tcs.2015.08.006.

**3** Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. *J. Funct. Program.*, 30:e14, 2020. `doi:10.1017/S095679682000012X`.

**4** Beniamino Accattoli and Giulio Guerrieri. Open call-by-value. In *Programming Languages and Systems*, volume abs/1609.00322, pages 206–226. Springer International Publishing, 2016. `doi:10.1007/978-3-319-47958-3_12`.

**5** Beniamino Accattoli and Giulio Guerrieri. Call-by-value solvability and multi types. *CoRR*, abs/2202.03079, 2022. `arXiv:2202.03079`.

**6** Beniamino Accattoli and Delia Kesner. The structural λ-calculus. In *Computer Science Logic*, pages 381–395. Springer Berlin Heidelberg, 2010. `doi:10.1007/978-3-642-15205-4_30`.

**7** Beniamino Accattoli and Luca Paolini. Call-by-value solvability, revisited. In *Functional and Logic Programming*, pages 4–16. Springer Berlin Heidelberg, 2012. `doi:10.1007/978-3-642-29822-6_4`.

**8** Henk Pieter Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. Elsevier, 1984. `doi:10.1016/c2009-0-14341-6`.

**9** Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. Not enough points is enough. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin Heidelberg, Lausanne, Switzerland, September 2007. `doi:10.1007/978-3-540-74915-8_24`.

**10** Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. The bang calculus revisited. In *Functional and Logic Programming*, pages 13–32. Springer International Publishing, 2020. `doi:10.1007/978-3-030-59025-3_2`.

**11** 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`.

**12** Alberto Carraro and Giulio Guerrieri. A semantical and operational account of call-by-value solvability. In *Lecture Notes in Computer Science*, volume 8412, pages 103–118. Springer Berlin Heidelberg, 2014. `doi:10.1007/978-3-642-54830-7_7`.

**13** M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für Mathematische Logik und Grundlagenforschung*, 19(1):139–156, December 1978. `doi:10.1007/bf02011875`.

**14** Daniel De Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, January 2017. `doi:10.1017/s0960129516000396`.

**15** José Espírito Santo. Delayed substitutions. In *Lecture Notes in Computer Science*, pages 169–183. Springer Berlin Heidelberg, 2007. `doi:10.1007/978-3-540-73449-9_14`.

**16** José Espírito Santo. The call-by-value lambda-calculus with generalized applications. In *28th EASCL Annual Conference on Computer Science Logic (CSL 2020)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2020. `doi:10.4230/LIPICS.CSL.2020.35`.

**17** José Espírito Santo, Maria João Frade, and Luís Pinto. Permutability in proof terms for intuitionistic sequent calculus with cuts. In Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2016.10`.

**18** José Espírito Santo, Delia Kesner, and Loïc Peyrot. A faithful and quantitative notion of distant reduction for generalized applications. In *Proc. of the 24th Int. Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, LNCS, April 2022. `arXiv:2201.04156`.

**19** José Espírito Santo and Luís Pinto. A calculus of multiary sequent terms. *ACM Transactions on Computational Logic*, 12(3):1–41, May 2011. `doi:10.1145/1929954.1929959`.

20 Álvaro García-Pérez and Pablo Nogueira. No solvable lambda-value term left behind. *Logical Methods in Computer Science*, 12(2), June 2016. doi:10.2168/lmcs-12(2:12)2016.

21 Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *Lecture Notes in Computer Science*, pages 555–574. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. doi:10.1007/3-540-57887-0_115.

22 Giulio Guerrieri, Luca Paolini, and Simona Ronchi Della Rocca. Standardization and conservativity of a refined call-by-value lambda-calculus. *Logical Methods in Computer Science ; Volume 13*, 2017. doi:10.23638/LMCS-13(4:29)2017.

23 Hugo Herbelin and Stéphane Zimmermann. An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, pages 142–156, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02273-9_12.

24 Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In *Rewriting Techniques and Applications*, pages 141–155. Springer Berlin Heidelberg, 2000. doi:10.1007/10721975_10.

25 Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003. doi:10.1007/s00153-002-0156-9.

26 Emma Kerinec, Giulio Manzonetto, and Michele Pagani. Revisiting Call-by-value Böhm trees in light of their Taylor expansion. *Logical Methods in Computer Science*, Volume 16, Issue 3, July 2020. doi:10.23638/LMCS-16(3:6)2020.

27 Delia Kesner. A theory of explicit substitutions with safe and full composition. *Log. Methods Comput. Sci.*, 5(3), 2009. arXiv:0905.2539.

28 Delia Kesner and Daniel Ventura. Quantitative types for the linear substitution calculus. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Lecture Notes in Computer Science*, pages 296–310, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-44602-7_23.

29 Delia Kesner and Andrés Viso. Encoding tight typing in a unified framework. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.27.

30 Ralph Matthes. Characterizing strongly normalizing terms for a lambda calculus with generalized applications via intersection types. In *Proc. ICALP 2000 (Geneva), volume 8 of Proceedings in Informatics*, pages 339–353, 2000.

31 Luca Paolini. Call-by-Value Separability and Computability. In Antonio Restivo, Simona Ronchi Della Rocca, and Luca Roversi, editors, *Theoretical Computer Science*, Lecture Notes in Computer Science, pages 74–89, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45446-2_5.

32 Luca Paolini and Simona Ronchi Della Rocca. Call-by-value solvability. *RAIRO - Theoretical Informatics and Applications*, 33(6):507–534, November 1999. doi:10.1051/ita:1999130.

33 Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

34 Neil Tennant. Ultimate normal forms for parallelized natural deductions. *Logic Journal of IGPL*, 10(3):299–337, 2002. doi:10.1093/jigpal/10.3.299.

35 Jan von Plato. Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567, 2001. doi:10.1007/s001530100091.

36 Christopher P. Wadsworth. The relation between computational and denotational properties for scott's $D_\infty$-models of the lambda-calculus. *SIAM Journal on Computing*, 5(3):488–521, September 1976. doi:10.1137/0205036.

# A Main proofs

In this section, we detail the crucial proofs showing that normalizability of a term $t$ w.r.t. the CBN/CBV solvable reductions implies that $t$ is CBN/CBV solvable. First, we define the number of head applications of a term as follows.

$$|x|_@ = 0 \qquad |\lambda x.t|_@ = |t|_@ \qquad |t(u, x.r)|_@ = \begin{cases} |r|_@ + |t|_@ + 1, & \text{if } x = \text{hv}(r) \\ |r|_@, & \text{otherwise} \end{cases}$$

Each proof will be decomposed in two statements. A first lemma states that solvable normal forms can be reduced to a value (surrounded by a distant context in the CBN case). The main property can then be shown by constructing an appropriate head context.

## A.1 Call-by-Name

The following intermediate lemma is stated with $\beta$-reduction, instead of $\mathtt{d}\beta$-reduction. This enables us to use it in the characterizations of both distant and non-distant CBN.

▶ **Lemma 42.** *For all $t = \mathtt{H}\langle\langle x\rangle\rangle \in \text{NF}_{\mathtt{sn}}$, $n \geq |t|_@$ and distant context $\mathtt{D}_0$, there is $m \geq 0$, there are variables $x_1, \ldots, x_m$ and distant contexts $\mathtt{D}, \mathtt{D}_1, \ldots \mathtt{D}_m$ such that $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t \to_\beta^* \mathtt{D}\langle \lambda x_m.\mathtt{D}_m\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|t|_@}\rangle\rangle\rangle$. In particular, if $t$ is neutral normal, then $m = 0$.*

**Proof.** By induction on $\langle |t|_@, t\rangle$. We reason by cases on the form of the normal term $t$.

- $t = x$, so that $|t|_@ = 0$ (this is the base case of the induction). We let $m = 0$, $\mathtt{D} = \mathtt{D}_0$ and conclude since $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}x = \mathtt{D}\langle \mathbf{o}^n\rangle = \mathtt{D}\langle \mathbf{o}^{n-|x|_@}\rangle$.
- $t = \lambda y.t'$, where $t' = \mathtt{H}'\langle\langle x\rangle\rangle$ $(x \neq y)$. We suppose w.l.o.g that $y \notin \text{fv}(\mathtt{D}_0\langle \mathbf{o}^n\rangle)$. Let $n \geq |t|_@ = |t'|_@$. By the *i.h.* there are $m'$, $x_1, \ldots, x_{m'}$ and $\mathtt{D}', \mathtt{D}_1, \ldots, \mathtt{D}_{m'}$ such that $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t' \to_\beta^* \mathtt{D}'\langle \lambda x_{m'}.\mathtt{D}_{m'}\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|t'|_@}\rangle\rangle\rangle$. Thus we obtain $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t \to_\beta^* \lambda y.\mathtt{D}'\langle \lambda x_{m'}.\mathtt{D}_{m'}\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|t'|_@}\rangle\rangle\rangle$ since $|t|_@ = |t'|_@$. We conclude by taking $m = m' + 1$, $x_m = y$, $\mathtt{D}_m = \mathtt{D}'$ and $\mathtt{D} = \Diamond$.
- $t = s(u, y.r)$, where $r = \mathtt{H}'\langle\langle y\rangle\rangle$ $(y \neq x)$ and $s = \mathtt{G}\langle\langle x\rangle\rangle$. We have $|t|_@ = |s|_@ + |r|_@ + 1$. Let $n \geq |t|_@ > |s|_@$. Applying the *i.h.* on the neutral normal term $s$, we know that for any $\mathtt{D}_0$, there is $\mathtt{D}'$ such that $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}s \to_\beta^* \mathtt{D}'\langle \mathbf{o}^{n-|s|_@}\rangle$.
  Let $n' = n - |s|_@ - 1$. Then $n' \geq |r|_@$ and we can show that $\{\mathtt{D}_0\langle \mathbf{o}^{n'+1}\rangle/x\}r \in \text{NF}_{\mathtt{sn}}$ and $|\{\mathtt{D}_0\langle \mathbf{o}^{n'+1}\rangle/x\}r|_@ = |r|_@$. Moreover, $\{\mathtt{D}_0\langle \mathbf{o}^{n'+1}\rangle/x\}r$ is of the form $\mathtt{H}''\langle\langle y\rangle\rangle$, for some $\mathtt{H}''$. We can then apply the *i.h.* on $\{\mathtt{D}_0\langle \mathbf{o}^{n'+1}\rangle/x\}r$, so there are $m', x_1, \ldots, x_{m'}, \mathtt{D}, \mathtt{D}_1, \ldots, \mathtt{D}_{m'}$ such that $\{\mathtt{D}'\langle \mathbf{o}^{n'}\rangle/y\}\{\mathtt{D}_0\langle \mathbf{o}^{n'+1}\rangle/x\}r \to_\beta^* \mathtt{D}\langle \lambda x_{m'}.\mathtt{D}_{m'}\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n'-|r|_@}\rangle\rangle\rangle$. We take $m = m'$. In the case where $t$ is neutral normal, we have $m = 0$ as required. Since $n' - |r|_@ = n - |t|_@$, we conclude as follows:

$$\begin{aligned}\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t &= \{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}s(\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}u, y.\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}r) \\ &\to_\beta^* \mathtt{D}'\langle \mathbf{o}^{n'+1}\rangle(\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}u, y.\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}r) \\ &\to_\beta \{\mathtt{D}'\langle \mathbf{o}^{n'}\rangle/y\}\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}r \to_\beta^* \mathtt{D}\langle \lambda x_m.\mathtt{D}_m\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|t|_@}\rangle\rangle\rangle.\end{aligned}$$

- $t = s(u, y.t')$, where $y \neq \text{hv}(t')$. Let $n \geq |t|_@ = |t'|_@$. By the *i.h.* on $t'$, for all $\mathtt{D}_0$ there are $m', x_1, \ldots, x_{m'}, \mathtt{D}', \mathtt{D}_1, \ldots, \mathtt{D}_{m'}$ s.t. $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t' \to_\beta^* \mathtt{D}'\langle \lambda x_{m'}.\mathtt{D}_{m'}\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|t'|_@}\rangle\rangle\rangle$. In particular $m' = 0$ if $t'$ is neutral normal. We set $\mathtt{D} = \{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}s(\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}u, y.\mathtt{D}')$ and $m = m'$. Since $|t'|_@ = |t|_@$, then $\{\mathtt{D}_0\langle \mathbf{o}^n\rangle/x\}t \to_\beta^* \mathtt{D}\langle \lambda x_m.\mathtt{D}_m\langle \ldots \lambda x_1.\mathtt{D}_1\langle \mathbf{o}^{n-|s|_@}\rangle\rangle\rangle$. ◀

▶ **Lemma 6.** *Let $t$ be an* $\texttt{sn}$*-normalizable term. Then $t$ is CBN solvable.*

**Proof.** Since $t$ is $\texttt{sn}$-normalizable, then there is a solvable normal term $t' \in \mathrm{NF}_{\texttt{sn}}$ such that $t \to^*_{\texttt{sn}} t'$ (and thus $t \to^*_{\texttt{dn}} t'$). Let $\mathrm{hv}(t') = x$. The term $t'$ can take two shapes:

1. $t' = \mathtt{H}'\langle\langle x\rangle\rangle$ if $x \in \mathrm{fv}(t')$;
2. $t' = \mathtt{D}_l\langle \lambda x_l.\dots\lambda x_2.\mathtt{D}_1\langle \lambda x_1.\mathtt{H}'\langle\langle x\rangle\rangle\rangle\rangle$ for $x \in \{x_1,\dots,x_l\}$, if $x \notin \mathrm{fv}(t')$.

In both cases, we must give a head context $\mathtt{H}$ such that $\mathtt{H}\langle t\rangle \to^*_{\texttt{dn}} \mathtt{D}\langle \mathtt{I}\rangle$ for a distant context $\mathtt{D}$.

We start with the first case ($x$ is free in $t'$). Let $n = |t'|_@$. By Lem. 42, there are $m \geq 0$, variables $y_1,\dots,y_m$ and distant contexts $\mathtt{D}',\mathtt{D}_1,\dots,\mathtt{D}_m$ such that $\{\mathbf{o}^n/x\}t' \to^*_\beta \mathtt{D}'\langle \lambda y_m.\mathtt{D}_m\langle \dots \lambda y_1.\mathtt{D}_1\langle \mathtt{I}\rangle\rangle\rangle$, which is also a $\texttt{dn}$-step. We let $\mathtt{H} = (\lambda x.\Diamond)(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m$. Then, we have:

$$\begin{aligned}\mathtt{H}\langle t\rangle &\to^*_{\texttt{dn}} \mathtt{H}\langle t'\rangle = (\lambda x.t')(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m \to_{\texttt{dn}} \{\mathbf{o}^n/x\}t'\overline{(\mathtt{I}, z.z)}^m \\ &\to^*_{\texttt{dn}} \mathtt{D}'\langle \lambda y_m.\mathtt{D}_m\langle \dots \lambda y_1.\mathtt{D}_1\langle \mathtt{I}\rangle\rangle\rangle\overline{(\mathtt{I}, z.z)}^m \to^m_{\texttt{dn}} \mathtt{D}'\langle\{\mathtt{I}/y_m\}\mathtt{D}_m\langle \dots\{\mathtt{I}/y_1\}\mathtt{D}_1\langle \mathtt{I}\rangle\rangle\rangle\end{aligned}$$

We conclude by taking $\mathtt{D} = \mathtt{D}'\langle\{\mathtt{I}/y_m\}\mathtt{D}_m\langle \dots\{\mathtt{I}/y_1\}\mathtt{D}_1\rangle\rangle$.

In the second case ($x$ is not free in $t'$), let $1 \leq i \leq l$ such that $x = x_i$. Let us consider the following reduction sequence:

$$t'\overline{(\mathtt{I}, z.z)}^{l-i} = \mathtt{D}_l\langle \lambda x_l.\dots\mathtt{D}_1\langle \lambda x_1.\mathtt{H}'\langle\langle x\rangle\rangle\rangle\rangle\overline{(\mathtt{I}, z.z)}^{l-i} \to^{l-i}_{\texttt{dn}} \mathtt{D}''\langle \lambda x.\mathtt{H}''\langle\langle x\rangle\rangle\rangle$$

where $\mathtt{D}'' = \mathtt{D}_l\langle\{\mathtt{I}/x_l\}\mathtt{D}_{l-1}\langle \dots\{\mathtt{I}/x_{i+1}\}\mathtt{D}_i\rangle\rangle$ and $\mathtt{H}'' = \{\mathtt{I}/x_j\}_{i<j\leq l}\mathtt{D}_{i-1}\langle \lambda x_{i-1}.\dots\mathtt{D}_1\langle \lambda x_1.\mathtt{H}'\rangle\rangle$. The subterm $\mathtt{H}''\langle\langle x\rangle\rangle$ above is obtained by substituting a $\texttt{sn}$-normal term with variables different from the head variable. This kind of substitution preserves the property of being $\texttt{sn}$-normal, so that $\mathtt{H}''\langle\langle x\rangle\rangle$ is $\texttt{sn}$-normal.

Let $n = |\mathtt{H}''\langle\langle x\rangle\rangle|_@$. Then Lem. 42 applied to $\{\mathbf{o}^n/x\}\mathtt{H}''\langle\langle x\rangle\rangle$ gives integers $m, y_1,\dots,y_m$ and distant contexts $\mathtt{D}',\mathtt{D}'_1,\dots,\mathtt{D}'_m$ such that (this is also a $\texttt{dn}$-step):

$$\{\mathbf{o}^n/x\}\mathtt{H}''\langle\langle x\rangle\rangle \to^*_\beta \mathtt{D}'\langle \lambda y_m.\mathtt{D}'_m\langle \dots \lambda y_1.\mathtt{D}'_1\langle \mathtt{I}\rangle\rangle\rangle.$$

To conclude, we let $\mathtt{H} = \Diamond\overline{(\mathtt{I}, z.z)}^{l-i}(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m$, where $m$ and $n$ were obtained before. The whole reduction from $\mathtt{H}\langle t\rangle$ goes as follows:

$$\begin{aligned}\mathtt{H}\langle t\rangle &\to^*_{\texttt{dn}} \mathtt{H}\langle t'\rangle = t'\overline{(\mathtt{I}, z.z)}^{l-i}(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m \\ &= \mathtt{D}_l\langle \lambda x_l.\dots\mathtt{D}_1\langle \lambda x_1.\mathtt{H}'\langle\langle x\rangle\rangle\rangle\rangle\overline{(\mathtt{I}, z.z)}^{l-i}(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m \\ &\to^{l-i}_{\texttt{dn}} \mathtt{D}''\langle \lambda x.\mathtt{H}''\langle\langle x\rangle\rangle\rangle(\mathbf{o}^n, z.z)\overline{(\mathtt{I}, z.z)}^m \\ &\to_{\texttt{dn}} \mathtt{D}''\langle\{\mathbf{o}^n/x\}\mathtt{H}''\langle\langle x\rangle\rangle\rangle\overline{(\mathtt{I}, z.z)}^m \\ &\to^*_{\texttt{dn}} \mathtt{D}''\langle \mathtt{D}'\langle \lambda y_m.\mathtt{D}'_m\langle \dots \lambda y_1.\mathtt{D}'_1\langle \mathtt{I}\rangle\rangle\rangle\rangle\overline{(\mathtt{I}, z.z)}^m \\ &\to^m_{\texttt{dn}} \mathtt{D}''\langle \mathtt{D}'\langle\{\mathtt{I}/y_m\}\mathtt{D}'_m\langle \dots\{\mathtt{I}/y_1\}\mathtt{D}'_1\langle \mathtt{I}\rangle\rangle\rangle\rangle\end{aligned}$$

where $\mathtt{D}'' = \mathtt{D}_l\langle\{\mathtt{I}/x_l\}\mathtt{D}_{l-1}\langle \dots\{\mathtt{I}/x_{i+1}\}\mathtt{D}_i\rangle\rangle$ and $\mathtt{H}'' = \{\mathtt{I}/x_j\}_{i<j\leq l}\mathtt{D}_{i-1}\langle \lambda x_{i-1}.\dots\mathtt{D}_1\langle \lambda x_1.\mathtt{H}'\rangle\rangle$.

We conclude by taking $\mathtt{D} = \mathtt{D}''\langle \mathtt{D}'\langle\{\mathtt{I}/y_m\}\mathtt{D}'_m\langle \dots\{\mathtt{I}/y_1\}\mathtt{D}'_1\rangle\rangle\rangle$ so that $\mathtt{H}\langle t\rangle \to^*_{\texttt{dn}} \mathtt{D}\langle \mathtt{I}\rangle$. ◀

## A.2 Call-by-Value

Before showing the property that $\texttt{sv}$-normalizable terms are CBV solvable, we need to show that $\texttt{pv}$-normalizable terms are potentially valuable. Here also, we use an intermediate lemma to prove that $\texttt{pv}$-nfs can be reduced to a value. The main difference between this lemma, as well as the one for $\texttt{sv}$, and the corresponding one in CBN, is that we must assign arbitrary terms $\mathbf{o}^n$ to free variables of $t$, and not just to the head variable. This lemma proceeds by induction on the grammar $\mathrm{NF}_{\texttt{pv}}$, we do not give the proof for lack of space.

▶ **Lemma 43.** *For all $t \in \mathrm{NF}_{\mathtt{pv}}$ with $\mathrm{fv}(t) \subseteq \{x_1, \ldots, x_m\}$, there exists $h \geq |t|_@$ such that for all $n_1, \ldots, n_m \geq h$ there exists a value $v$ such that $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t \rightarrow^*_{\beta_\mathtt{v}} v$. If $t \in \mathrm{NE}_{\mathtt{pv}}$ with $\mathrm{hv}(t) = x_i$ (necessarily free), then $v = \mathbf{o}^{n_i - |t|_@}$.*

▶ **Lemma 21.** *Let $t$ be a $\mathtt{pv}$-normalizable term. Then $t$ is potentially valuable.*

**Proof.** Since $t$ is $\mathtt{pv}$-normalizable, then there is a $\mathtt{pv}$-normal term $t'$ such that $t \rightarrow^*_{\mathtt{pv}} t'$. Therefore $t' \in \mathrm{NF}_{\mathtt{pv}}$ by Lem. 20. Let $\mathrm{fv}(t) = \{x_1, \ldots, x_m\}$, so that $\mathrm{fv}(t') \subseteq \{x_1, \ldots, x_m\}$. By Lem. 43, there is $h \geq |t'|_@$ such that $\{\mathbf{o}^h/x_m\} \ldots \{\mathbf{o}^h/x_1\}t' \rightarrow^*_{\beta_\mathtt{v}} v$ for some value $v$. Consider $\mathtt{D} = \mathtt{I}(\mathbf{o}^h, x_1.\mathtt{I}(\mathbf{o}^h, x_2. \ldots \mathtt{I}(\mathbf{o}^h, x_m.\Diamond) \ldots))$. Then,

$$\begin{aligned} \mathtt{D}\langle t\rangle &\rightarrow^*_{\mathtt{pv}} \mathtt{I}(\mathbf{o}^h, x_1. \ldots \mathtt{I}(\mathbf{o}^h, x_m.t')) \rightarrow_{\beta_\mathtt{v}} \mathtt{I}(\mathbf{o}^h, x_2. \ldots \mathtt{I}(\mathbf{o}^h, x_m.\{\mathbf{o}^h/x_1\}t') \ldots) \\ &\rightarrow^*_{\beta_\mathtt{v}} \{\mathbf{o}^h/x_m\} \ldots \{\mathbf{o}^h/x_1\}t' \rightarrow^*_{\beta_\mathtt{v}} v \text{ (by Lem. 43)} \end{aligned}$$

As a consequence, $\mathtt{D}\langle t\rangle \rightarrow^*_{\mathtt{dv}} v$. ◀

We are now ready to prove the property for $\mathtt{sv}$-reduction. First, the intermediate Lemma.

▶ **Lemma 44.** *For all $t \in \mathrm{NF}_{\mathtt{sv}}$ with $\mathrm{fv}(t) \subseteq \{x_1, \ldots, x_m\}$, there exist $h \geq |t|_@, k \geq 0$ such that for all $n_1, \ldots, n_{m+k} \geq h$ there exists $n \geq 0$ such that the following holds $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k}}, z.z) \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^n$.*

**Proof.** By induction on $t \in \mathrm{NF}_{\mathtt{sv}}$.

- $t$ is a variable, thus $t = x_i$. We take $h = 0 = |x_i|_@, k = 0$ so that for all $n_1, \ldots, n_m \geq 0$ we have $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t = \mathbf{o}^{n_i}$. We let $n = n_i \geq 0$ and we conclude.
- $t = \lambda x.s$ with $s \in \mathrm{NF}_{\mathtt{sv}}$. We suppose w.l.o.g that $x \notin \{x_1, \ldots, x_m\}$. Then, $\mathrm{fv}(s) \subseteq \{x, x_1, \ldots, x_m\}$. By the *i.h.*, there exist $h' \geq |s|_@ = |t|_@, k' \geq 0$ such that for all $n', n_1, \ldots, n_{m+k} \geq h'$ there exists $n \geq 0$ such that

  $$\{\mathbf{o}^{n'}/x\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}s(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^n .$$

  Taking $h = h'$ and $k = k' + 1$ we have:

  $$\begin{aligned} &\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t(\mathbf{o}^{n'}, \mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \\ &= \lambda x.\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}s(\mathbf{o}^{n'}, \mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \\ &\rightarrow_{\beta_\mathtt{v}} \{\mathbf{o}^{n'} \backslash\!\!\backslash x\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}s(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \\ &= \{\mathbf{o}^{n'}/x\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}s(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^n \text{ (by the } i.h.) \end{aligned}$$

- $t = s(u, y.r)$ with $s \in \mathrm{NE}_{\mathtt{pv}}$, $u \in \mathrm{NF}_{\mathtt{pv}}$ and $r \in \mathrm{NF}_{\mathtt{sv}}$. We suppose w.l.o.g that $y \notin \{x_1, \ldots, x_m\}$. Thus $\mathrm{fv}(r) \subseteq \{y, x_1, \ldots, x_m\}$. Let $x_j = \mathrm{hv}(s)$ for some $1 \leq j \leq m$. By Lem. 43 and the *i.h.*respectively:
  1. There is $h_s \geq |s|_@$ s.t. for all $n^s_1, \ldots, n^s_m \geq h_s$ we have $\{\mathbf{o}^{n^s_m}/x_m\} \ldots \{\mathbf{o}^{n^s_1}/x_1\}s \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^{n^s_j - |s|_@}$.
  2. There is $h_u \geq |u|_@$ such that for all $n^u_1, \ldots, n^u_m \geq h_u$ there is a value $v$ such that $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}u \rightarrow^*_{\beta_\mathtt{v}} v$.
  3. There are $h_r \geq |r|_@, k' \geq 0$ such that for all $n_y, n^r_1, \ldots, n^r_{m+k'} \geq h_r$ there is $n \geq 0$ such that $\{\mathbf{o}^{n_y}/y\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}r(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^n$.

  We take $h = \max(h_s + h_r + 1, h_u) \geq |t|_@$ and we consider any $n_1, \ldots, n_m \geq h$.
  - We have $h \geq h_s + h_r + 1$ and thus $n_1, \ldots, n_m \geq h$ implies in particular $n_1, \ldots, n_m \geq h_s$. This gives $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}s \rightarrow^*_{\beta_\mathtt{v}} \mathbf{o}^{n_j - |s|_@}$ by (1).

- We have $h \geq h_u$ and thus $n_1, \ldots, n_m \geq h$ implies in particular $n_1, \ldots, n_m \geq h_u$. This gives $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}u \to^*_{\mathtt{pv}} v$ by (2).
- We have $h \geq h_r + 1 > h_r$ and thus $n_1, \ldots, n_m \geq h$ implies in particular $n_1, \ldots, n_m \geq h_r + h_s + 1 \geq h_r + |s|_@ + 1 > h_r$. This gives $n \geq 0$ such that by the *i.h.* (3) $\{\mathbf{o}^{n_j - |s|_@ - 1}/y\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}r(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k'}}, z.z) \to^*_{\beta_\mathtt{v}} \mathbf{o}^n$.

In summary, we reduce as follows:

$$\begin{aligned}
&\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k}}, z.z) \\
&\to^*_{\beta_\mathtt{v}} \mathbf{o}^{n_j - |s|_@}(v, y.\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}r)(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k}}, z.z) \\
&\to_{\beta_\mathtt{v}} \{\mathbf{o}^{n_j - |s|_@ - 1}/y\}\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}r(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k}}, z.z) \\
&\to^*_{\beta_\mathtt{v}} \mathbf{o}^n \quad \text{(by the } i.h. \text{ (3))}
\end{aligned}$$

◀

▶ **Lemma 24.** *Let $t$ be an* `sv`*-normalizable term. Then $t$ is CBV solvable.*

**Proof.** Since $t$ is `sv`-normalizable, then there is an `sv`-normal term $t'$ such that $t \to^*_{\mathtt{sv}} t'$. Therefore $t' \in \mathrm{NF}_{\mathtt{sv}}$ by Lem. 23. Let $\mathrm{fv}(t) = \{x_1, \ldots, x_m\}$, so that $\mathrm{fv}(t') \subseteq \{x_1, \ldots, x_m\}$. By Lem. 42, there are $h, k \in \mathbb{N}$ such that for all $n_1, \ldots, n_{m+k} \geq h$ there is $n \geq 0$ such that $\{\mathbf{o}^{n_m}/x_m\} \ldots \{\mathbf{o}^{n_1}/x_1\}t'(\mathbf{o}^{n_{m+1}}, \ldots, \mathbf{o}^{n_{m+k}}, z.z) \to^*_{\beta_\mathtt{v}} \mathbf{o}^n$, which is also a `dv`-step. We take $n_1, \ldots, n_{m+k} = h$. We can then write $(\mathbf{o}^h, \ldots, \mathbf{o}^h, z.z)$ as $\overline{(\mathbf{o}^h, z.z)}^k$. Let $\mathtt{H} = \mathtt{I}(\mathbf{o}^h, x_m. \ldots \mathtt{I}(\mathbf{o}^h, x_1.\Diamond) \ldots)\overline{(\mathbf{o}^h, z.z)}^k\overline{(\mathtt{I}, z.z)}^n$. Then:

$$\mathtt{H}\langle t\rangle \to^*_{\mathtt{sv}} \mathtt{H}\langle t'\rangle \to^m_{\beta_\mathtt{v}} \mathtt{dv}\{\mathbf{o}^h/x_m\} \ldots \{\mathbf{o}^h/x_1\}t'\overline{(\mathbf{o}^h, z.z)}^k\overline{(\mathtt{I}, z.z)}^n \to^*_{\beta_\mathtt{v}} \mathbf{o}^n\,\overline{(\mathtt{I}, z.z)}^n \to^n_{\beta_\mathtt{v}} \mathtt{I}$$

As a consequence, $\mathtt{H}\langle t\rangle \to_{\mathtt{dv}} \mathtt{I}$. ◀

# B Quantitative Type Systems for λ-calculi with Explicit Substitutions

We use the following type system for the proofs of equivalence between our calculi and the λ-calculus with explicit substitutions in Sec. 6.

We start with the **CBN type system** of [28]. The grammar of types is the same as ours for the CBN system. This system differs, in the presentation only, by the fact that we write (MANY) as a separate rule.

$$\frac{}{x : [\sigma] \vdash x : \sigma}\ (\textsc{ax}) \qquad \frac{(\Gamma_i \vdash M : \sigma_i)}{\wedge_{i\in I}\Gamma_i \vdash M : [\sigma_i]_{i\in I}}\ (\textsc{many}) \qquad \frac{\Gamma; x : \mathcal{M} \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \mathcal{M} \to \sigma}\ (\to_i)$$

$$\frac{\Gamma \vdash M : \mathcal{M} \to \sigma \qquad \Delta \vdash N : \mathcal{M}}{\Gamma \wedge \Delta \vdash MN : \sigma}\ (\to_e) \qquad \frac{\Gamma; x : \mathcal{M} \vdash M : \sigma \qquad \Delta \vdash N : \mathcal{M}}{\Gamma \wedge \Delta \vdash [N/x]M : \sigma}\ (\textsc{cut})$$

For the **CBV type system**, we define a new type system using the rules of the system from [5], but using our CBV type grammar. A difference in the presentation is that we include rule (MANY) inside the rules for variables and abstractions. Typability is equivalent in both systems.

$$\frac{}{x : \mathcal{M} \vdash x : \mathcal{M}}\ (\textsc{ax}) \qquad \frac{(\Gamma_i; x : \mathcal{M}_i \vdash M : \sigma_i)_{i\in I}}{\wedge_{i\in I}\Gamma_i \vdash \lambda x.M : [\mathcal{M}_i \to \sigma_i]_{i\in I}}\ (\lambda)$$

$$\frac{\Gamma \vdash M : [\mathcal{M} \to \mathcal{N}] \qquad \Delta \vdash N : \mathcal{M}}{\Gamma \wedge \Delta \vdash MN : \mathcal{N}}\ (@) \qquad \frac{\Gamma; x : \mathcal{M} \vdash M : \sigma \qquad \Delta \vdash N : \mathcal{M}}{\Gamma \wedge \Delta \vdash [N/x]M : \sigma}\ (\textsc{es})$$

# Normalization Without Syntax

**Willem B. Heijltjes**
Department of Computer Science, University of Bath, UK

**Dominic J. D. Hughes**
Logic Group, University of California Berkeley, CA, USA

**Lutz Straßburger**
Equipe Partout, Inria Saclay, Palaiseau, France

## Abstract

We present normalization for intuitionistic combinatorial proofs (ICPs) and relate it to the simply-typed lambda-calculus. We prove confluence and strong normalization. Combinatorial proofs, or "proofs without syntax", form a graphical semantics of proof in various logics that is canonical yet complexity-aware: they are a polynomial-sized representation of sequent proofs that factors out exactly the non-duplicating permutations. Our approach to normalization aligns with these characteristics: it is canonical (free of permutations) and generic (readily applied to other logics). Our reduction mechanism is a canonical representation of reduction in sequent calculus with closed cuts (no abstraction is allowed below a cut), and relates to closed reduction in lambda-calculus and supercombinators. While we will use ICPs concretely, the notion of reduction is completely abstract, and can be specialized to give a reduction mechanism for any representation of typed normal forms.

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

**Keywords and phrases** combinatorial proofs, intuitionistic logic, lambda-calculus, Curry–Howard, proof nets

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.19

**Related Version** *Technical report*: `https://hal.inria.fr/hal-03654060` [17]

**Funding** Supported by EPSRC Grant EP/R029121/1 *Typed Lambda-Calculi with Sharing and Unsharing* and Inria Associated Team *Combinatorial Proof Normalization (COMPRONOM)*.

**Acknowledgements** Dominic Hughes would like to thank Wes Holliday, his host at U.C. Berkeley. We thank the referees for their diligence.

## 1 Introduction

The sequent calculus was introduced by Gentzen [9] as a meta-calculus, to describe the construction of proofs in natural deduction, the object-calculus. The sequent calculus has good proof-theoretic properties, such as isolating the cut-rule as the distinction between normal and non-normal proofs and avoiding the ad-hoc construction of open and closed assumptions. However, it features many permutations, that relate different ways of constructing the same natural deduction proof. This is a problem for proof normalization in particular, since permutations come to dominate the cut-elimination process.

When Girard introduced Linear Logic [10], it was naturally expressed in sequent calculus, which defined clear and natural meta-level operations for proof construction. But there was no object-level calculus to which these applied, and which might capture its computational content. Constructing one became the project of *proof nets* [10, 12, 20, 15], with the aim of *canonicity*: proof nets aim to represent sequent proofs canonically, modulo permutations.

© Willem B. Heijltjes, Dominic J. D. Hughes, and Lutz Straßburger;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 19; pp. 19:1–19:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Combinatorial proofs, first developed for classical propositional logic by Hughes [18], continue the tradition of proof nets with a refined aim, called *local canonicity* [19]. The issue is that permutations may *duplicate* subproofs; to factor them out then generally causes an exponential blowup of the representation. Figure 1 illustrates such a permutation. The idea of *local canonicity* is to give a complexity-sensitive, polynomial representation of sequent proofs, modulo the non-duplicating permutations. This is achieved in combinatorial proofs by a clean separation of the logical content (the logical rules of a sequent proof) and the structural content (the structural rules, contraction and weakening), each captured in a distinct part of a combinatorial proof. Sequent calculi are generally unable to stratify proofs in this way, but it is a natural form of decomposition in deep inference [30]. Beyond classical propositional logic, combinatorial proofs have been given for intuitionistic propositional logic [16], first-order classical logic [21, 22], relevance logics [2], and modal logics [3].

The problem of exponential duplication appears also at the level of formula isomorphisms [6, 8], and is usefully illustrated there. The formula-isomorphisms of symmetry, associativity, and currying, below, do not affect the size of the formula.

$$A \wedge B \;\sim\; B \wedge A \qquad A \wedge (B \wedge C) \;\sim\; (A \wedge B) \wedge C \qquad (A \wedge B) \Rightarrow C \;\sim\; A \Rightarrow (B \Rightarrow C)$$

But the distributivity isomorphism, below, duplicates the antecedent of an implication, and its repeated application may cause exponential growth. Combinatorial proofs, as a complexity-aware graphical formalism, factor out the former three, but not the latter.

$$A \Rightarrow (B \wedge C) \;\sim\; (A \Rightarrow B) \wedge (A \Rightarrow C)$$

We are interested in the question: what is a natural and general notion of composition for combinatorial proofs? In this paper we consider the intuitionistic case – *Intuitionistic Combinatorial Proofs* (ICPs) [16] – where the question is particularly pertinent due to the Curry–Howard correspondence with typed lambda-calculi.

Our aim has been twofold: 1) to implement sequent-calculus reduction canonically (i.e. without permutations), and 2) to ensure our notion of reduction is sufficiently abstract that it will (plausibly) generalize to combinatorial proofs more widely.

Our solution is a notion of composition in conjunction-implication intuitionistic logic that is locally canonical for sequent calculus normalization, in the sense that non-duplicating permutations on cuts are factored out. Reduction operates on trees of normal forms, where edges represent cuts, giving a simple and natural structure that may easily generalize to other logics. A reduction step on a given edge is determined by how the attached nodes may sequentialize, not by their internal structure. Consequently, the reduction mechanism is *abstract* in the sense that it is agnostic about the actual contents of nodes, which can be any representation of normal forms. Beyond the scope of this paper, the mechanism generalizes straightforwardly to classical logic, which we will briefly expand on in the conclusion.

Proofs are omitted; a version with all proofs in an appendix is on the HAL archive [17].

## 1.1 Composition

Composition of proofs in intuitionistic sequent calculus is by the following cut-rule, followed by cut-elimination. We would like to transport this operation to combinatorial proofs.

$$\frac{\Gamma \vdash A \qquad A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\,\mathsf{cut}$$

We identify two prominent approaches for similar composition operations in the literature (our classification is not intended to be comprehensive, only helpful in setting out similarities):

$$\dfrac{\Gamma \vdash A \qquad \dfrac{B, B, \Delta \vdash C}{B, \Delta \vdash C}\,\mathsf{c}}{\Gamma, A \Rightarrow B, \Delta \vdash C}\Rightarrow\mathsf{L} \quad \approx \quad \dfrac{\dfrac{\Gamma \vdash A \qquad \dfrac{\Gamma \vdash A \quad B, B, \Delta \vdash C}{B, \Gamma, A \Rightarrow B, \Delta \vdash C}\Rightarrow\mathsf{L}}{\Gamma, A \Rightarrow B, \Gamma, A \Rightarrow B, \Delta \vdash C}\Rightarrow\mathsf{L}}{\Gamma, A \Rightarrow B, \Delta \vdash C}\,\mathsf{c}$$

**Figure 1 A duplicating permutation.** Intuitionistic sequent calculus, as we will use it, has exactly one duplicating permutation, illustrated here. Permuting the contraction rule $\mathsf{c}$ and the implication-left rule $\Rightarrow\mathsf{L}$ duplicates the subproof on the left. Iterating the permutation gives exponential growth. It is instructive to consider the translation to natural deduction, which unfolds along this permutation and does indeed grow exponentially.

**Internal rewriting.** An object-calculus may support non-normal forms and rewriting internally. In the $\lambda$-calculus, composition creates a redex, which is then beta-reduced. Likewise, many notions of proof net admit an explicit notion of cut, as a node or as a *cut-link* connecting dual formulae, that is eliminated by rewriting [12, 19], giving rise to the interaction nets paradigm [27].

**Direct composition.** For an object calculus that admits only normal forms, composition may be computed by a single-shot operation. Examples are the Geometry of Interaction, which computes a normal form via the execution formula [11]; game semantics, which composes strategies by *interaction + hiding* [1, 25]; evaluation of cut-nets in ludics [13]; and various notions of proof net where composition is a form of path composition over links [20, 15, 23].

Observe that object-level proofs become an invariant for sequent-calculus cut-elimination. Based on prior art, one may readily imagine what either approach would involve for ICPs. For internal rewriting, an ICP may be constructed over a sequent that includes internal cut-formulas as special antecedents $A \Rightarrow A$ (marked below by underlining), introduced by a cut as analogous to a $\Rightarrow\mathsf{L}$ rule, and eliminated by rewriting. One may transport sequent-calculus cut-elimination to this setting by identifying sub-proofs of ICPs, via *kingdoms* [4].

$$\Gamma, \underline{A_1 \Rightarrow A_1}, \ldots, \underline{A_n \Rightarrow A_n} \vdash B$$

For direct composition, ICPs may be interpreted as games with *sharing* [16], for which the *interaction + hiding* approach can be explored. Both these approaches are interesting and deserve to be investigated, and we may do so in future. However, they will inevitably require some intricate combinatorics, and are not likely to generalize across combinatorial proofs.

Here, we describe a normalization method for ICPs that is simple, natural, and achieves both our main objectives: 1) it is effectively a permutation-free implementation of sequent calculus cut-elimination, and 2) it is sufficiently abstract that it is likely to generalize well. Technically, ICPs will form the nodes of a *combinatorial tree*, connected by edges that represent cuts. Combinatorial trees are then reduced by cut-elimination, following the reduction in sequent calculus. Interestingly, this approach fits neither of the above categories well, and instead suggests to identify a third category:

**External rewriting.** An object calculus without internal composition may be extended by a secondary structure, which is then evaluated by rewriting. The prime example is *supercombinators* [24, 29], where normalization takes place on a tree of normal-form $\lambda$-terms (restricted to having no abstractions inside applications).

We explore the parallels between our combinatorial trees and supercombinators in Section 7. In addition, we connect ICP normalization to *closed reduction* in $\lambda$-calculus [7] in Section 8, via a novel explicit-substitution calculus, the *combinatory $\lambda$-calculus*, in Section 6.

## 2 Intuitionistic Combinatorial Proofs

We give a concise inductive definition of ICPs; see [16] for a full treatment including an informal introduction and a geometric definition. For the purposes of this paper, it would also be sufficient to view ICPs as sequent proofs modulo permutations.

We work in conjunction–implication intuitionistic logic. ***Formulas*** $A, B, C$ are given by the grammar below, where $P, Q$ are propositional atoms. A ***context*** $\Gamma, \Delta$ is a multiset of formulas and a ***sequent*** $\Gamma \vdash A$ is a context with a formula.

$$A, B, C ::= P \mid A \wedge B \mid A \Rightarrow B$$

An ICP for a formula $A$ will be a graph homomorphism $f : \mathcal{G} \to [\![A]\!]$ consisting of:

- an ***arena*** $[\![A]\!]$, a graph representing the formula $A$ modulo the non-duplicating isomorphisms of symmetry, associativity, and currying;
- a ***linked arena*** $\mathcal{G}$, a proof net in IMLL (intuitionistic multiplicative linear logic) over an arena rather than a formula, to represent the *logical* rules of the sequent calculus;
- a ***skew fibration*** $f$, a graph homomorphism from $\mathcal{G}$ to $[\![A]\!]$ representing the *structural* rules of contraction and weakening.

We define each component inductively. An arena will be a DAG (directed acyclic graph) $\mathcal{G} = (V_\mathcal{G}, \twoheadrightarrow_\mathcal{G})$ with vertices $V_\mathcal{G}$ and edges $\twoheadrightarrow_\mathcal{G} \subseteq V_\mathcal{G} \times V_\mathcal{G}$. We indicate the ***root vertices*** of $\mathcal{G}$ (those without outgoing edges) by $R_\mathcal{G}$. Consider the following two operations: a ***sum*** of two graphs $\mathcal{G} + \mathcal{H}$ is their disjoint union, and a ***subjunction*** $\mathcal{G} \triangleright \mathcal{H}$ is a disjoint union that in addition connects all the roots of $\mathcal{G}$ to the roots of $\mathcal{H}$.

$$\begin{array}{lrcl} \text{sum:} & \mathcal{G} + \mathcal{H} & = & (V_\mathcal{G} \uplus V_\mathcal{H}, \twoheadrightarrow_\mathcal{G} \uplus \twoheadrightarrow_\mathcal{H}) \\ \text{subjunction:} & \mathcal{G} \triangleright \mathcal{H} & = & (V_\mathcal{G} \uplus V_\mathcal{H}, \twoheadrightarrow_\mathcal{G} \uplus \twoheadrightarrow_\mathcal{H} \uplus (R_\mathcal{G} \times R_\mathcal{H})) \end{array}$$

▶ **Definition 1.** *An* ***arena*** *is a graph $\mathcal{G}$ constructed from single vertices by $(+)$ and $(\triangleright)$, with an $L$-**labelling** $\ell_\mathcal{G} : V_\mathcal{G} \to L$ assigning each vertex a label from a set $L$. The arena $[\![A]\!]$ of a formula $A$ is given inductively as follows: $[\![P]\!]$ is a single vertex labelled $P$, and*

$$[\![A \wedge B]\!] = [\![A]\!] + [\![B]\!] \qquad \text{and} \qquad [\![A \Rightarrow B]\!] = [\![A]\!] \triangleright [\![B]\!] \ .$$

Note that arenas are linear in the size of formulas, and while they factor out symmetry, associativity, and currying, they do not factor out distributivity.

$$[\![A \Rightarrow (B \wedge C)]\!] \neq [\![(A \Rightarrow B) \wedge (A \Rightarrow C)]\!]$$

An ICP will be an *arena morphism*: a map $f : \mathcal{G} \to [\![A]\!]$ given by an underlying function on vertices $f : V_\mathcal{G} \to V_{[\![A]\!]}$ that preserves edges, roots, and the equivalence given by labelling, i.e. if $\ell_\mathcal{G}(v) = \ell_\mathcal{G}(w)$ then $\ell_{[\![A]\!]}(f(v)) = \ell_{[\![A]\!]}(f(w))$. We will construct arena morphisms inductively, which guarantees these conditions. For $g : \mathcal{G} \to [\![A]\!]$ and $h : \mathcal{H} \to [\![B]\!]$ we have the operations

$$\begin{array}{lrcll} \text{implication:} & g \triangleright h & : & \mathcal{G} \triangleright \mathcal{H} \to [\![A]\!] \triangleright [\![B]\!] & \\ \text{sum:} & g + h & : & \mathcal{G} + \mathcal{H} \to [\![A]\!] + [\![B]\!] & \\ \text{contraction:} & [g, h] & : & \mathcal{G} + \mathcal{H} \to [\![A]\!] & \text{(where } [\![A]\!] = [\![B]\!]) \end{array}$$

where each case is given by the union of the underlying functions on vertex sets: for implication and sum, $g \cup h : (V_\mathcal{G} \uplus V_\mathcal{H}) \to (V_{[\![A]\!]} \uplus V_{[\![B]\!]})$, and for contraction $g \cup h : (V_\mathcal{G} \uplus V_\mathcal{H}) \to V_{[\![A]\!]}$. In addition, we use the following constructions, where $\varnothing$ is the empty graph.

$$\begin{array}{lrcl} \text{axiom:} & 1_{P,Q} & : & [\![P]\!] \to [\![Q]\!] \\ \text{weakening:} & \varnothing_A & : & \varnothing \to [\![A]\!] \end{array}$$

$$\frac{}{1 :: P \vdash 1 :: P}\ \mathsf{ax}^{*} \qquad \frac{\varphi :: \Gamma \vdash f :: B}{\varphi :: \Gamma, \varnothing :: A \vdash f :: B}\ \mathsf{w} \qquad \frac{\varphi :: \Gamma, k :: A, l :: A \vdash f :: B}{\varphi :: \Gamma, [k, l] :: A \vdash f :: B}\ \mathsf{c}^{\dagger}$$

$$\frac{\varphi :: \Gamma, k :: A, l :: B \vdash f :: C}{\varphi :: \Gamma, k{+}l :: A \wedge B \vdash f :: C}\ \wedge\mathsf{L} \qquad \frac{\varphi :: \Gamma \vdash f :: A \qquad \psi :: \Delta \vdash g :: B}{\varphi :: \Gamma, \psi :: \Delta \vdash f{+}g :: A \wedge B}\ \wedge\mathsf{R}$$

$$\frac{\varphi :: \Gamma, k :: A \vdash f :: B}{\varphi :: \Gamma \vdash k \triangleright f :: A \Rightarrow B}\ {\Rightarrow}\mathsf{R} \qquad \frac{\varphi :: \Gamma \vdash f :: A \qquad k :: B, \psi :: \Delta \vdash g :: C}{\varphi :: \Gamma, f \triangleright k :: A \Rightarrow B, \psi :: \Delta \vdash g :: C}\ {\Rightarrow}\mathsf{L}^{\ddagger}$$

**Figure 2** Inductive construction of ICPs. ($*$) Each instance of ax is given a distinct label in the source arena. (†) For c we require $k, l \neq \varnothing$. (‡) For ⇒L we require $k \neq \varnothing$.

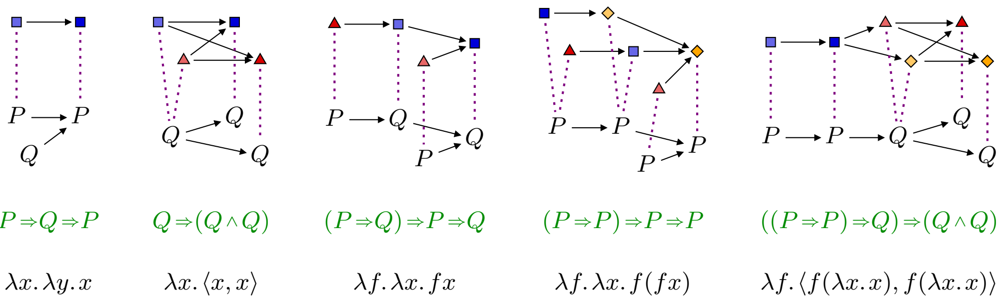



**Figure 3** Examples of ICPs with corresponding $\lambda$-terms. The source arena is at the top, with its labelling given by coloured shapes. The target arena is at the bottom, labelled with propositional atoms, and the arena morphism is given by dotted (purple) lines.

The axiom is the trivial map from one singleton arena (with vertex labelled $P$) to another (with vertex labelled $Q$). Weakening is the empty morphism. Note that because arenas are non-empty, weakening in isolation is not an arena morphism, but we will use it only in the context of an implication, sum, or contraction, so that this is not an issue.

We write $f :: A$ for $f : \mathcal{G} \to [\![A]\!]$. To construct ICPs from sequent proofs we use ***sequents*** of arena morphisms (and weakenings), that represent a single arena morphism as follows.

$$k_1 :: A_1, \ldots, k_n :: A_n \vdash f :: B \quad \Longleftrightarrow \quad (k_1 + \ldots + k_n) \triangleright f \ :: \ (A_1 \wedge \ldots \wedge A_n) \Rightarrow B$$

We refer to $f$ and the $k_i$ as ***ports***, where $k_i$ is an ***antecedent*** and $f$ the ***consequent***, and we write $\varphi :: \Gamma$ for the ***context*** $k_1 :: A_1, \ldots, k_n :: A_n$.

▶ **Definition 2.** *An **intuitionistic combinatorial proof (ICP)** of a formula $A$ is an arena morphism $f :: A$ constructed by the sequent calculus of Figure 2.*

Figure 3 gives examples of ICPs, with corresponding types and $\lambda$-terms (the translation will be made formal in Section 8). Figure 4 gives non-examples of ICPs.

For clarity, an axiom ax generates the ICP below.

$$1 :: P \vdash 1 :: P \quad = \quad \begin{matrix} \blacksquare \longrightarrow \blacksquare \\ \vdots \qquad \vdots \\ P \longrightarrow P \end{matrix}$$

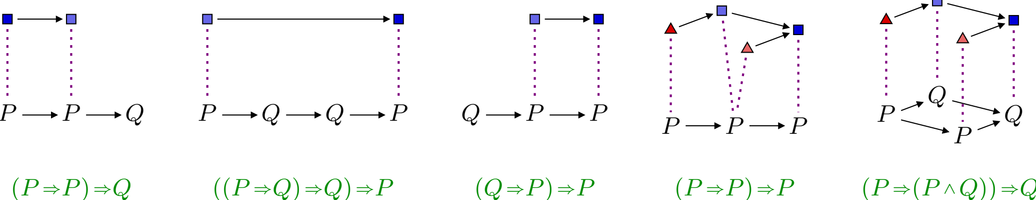



**Figure 4** Non-examples of ICPs. They cannot be constructed with the sequent calculus in Figure 2.

**a)**
$$\frac{\Gamma \vdash A \qquad A, \Delta \vdash B}{\Gamma, \Delta \vdash B}\ \mathsf{cut}$$

**b)**
$$\varphi :: \Gamma \vdash f :: A \quad = \quad \frac{\varphi :: \Gamma}{f :: A}$$
$$k :: A, \psi :: \Delta \vdash g :: B \quad = \quad \frac{k :: A \quad \psi :: \Delta}{g :: B}$$

**c)**
$$\begin{array}{c} \varphi :: \Gamma \\ \hline f :: A \\ \text{- - - - -} \\ k :: A \quad \psi :: \Delta \\ \hline g :: B \end{array}$$

**Figure 5** Composition of combinatorial proofs into combinatorial trees. **a)** The sequent calculus cut-rule. **b)** Presenting ICP sequents as nodes of a tree, with antecedent ports above and consequent port below a central line. **c)** Connecting both nodes by an edge, represented by a dashed line, to form a tree.

We call the subgraph ▪→▪ a ***link***, where the side condition ($\star$) in Figure 2 requires that every link receives a different label ■, ▲, ◆, etc. Vertices are ***equivalent*** if they have the same label, and ICPs as arena morphisms preserve equivalence by construction.

To *decompose* an ICP, the unary rules $\wedge\mathsf{L}, \Rightarrow\mathsf{R}, \mathsf{c}, \mathsf{w}$ apply whenever the given port is of the right kind, respectively $k+l$, $k \triangleright f$, $[k,l]$, and $\varnothing$. The binary rules $\wedge\mathsf{R}, \Rightarrow\mathsf{L}$ apply only when the ICP can be split into two without breaking up any links in the source graph. We write $\varphi \mid\mid \psi$ when the sources of $\varphi$ and $\psi$ do not share any labels; then the rules $\wedge\mathsf{R}, \Rightarrow\mathsf{L}$ as given in Figure 2 apply in reverse exactly when respectively $\varphi, f \mid\mid \psi, g$ and $\varphi, f \mid\mid k, \psi, g$. We call a port ***open*** if the ICP can be decomposed along it, and ***closed*** otherwise.

We refer to [16] for a *geometric* definition of ICPs, where the equivalence with the *inductive* definition given here is a theorem. We recall the following from [16].

▶ **Theorem 3** (Local canonicity). *Two sequent proofs construct the same ICP if and only if they are equivalent modulo non-duplicating rule permutations and formula-isomorphisms.*

## 3 Composition of combinatorial proofs

Combinatorial proofs represent normal forms: the sequent calculus for constructing them, in Figure 2, does not have a cut-rule (Figure 5a). What is expected is a notion of composition, of an ICP for $\Gamma \vdash A$ and one for $A, \Delta \vdash B$ into one for $\Gamma, \Delta \vdash B$.

We give a direct interpretation of composition by taking ICPs as the nodes of a tree, connected by cuts as edges; see Figure 5, where solid lines represent the nodes in the tree and the dashed lines the edges. We formalize this construction as a notion of ***combinatorial tree***, which we will then proceed to reduce. The nature of reduction will make it desirable to have constants available.

$$\dfrac{\dfrac{t}{1::P}}{1::P} \xrightarrow{[1]} t \qquad \dfrac{\dfrac{\tau}{\varphi}\quad \dfrac{s}{[k,l]::A}}{f::B} \xrightarrow[(k,l\neq\varnothing)]{[c]} \dfrac{\dfrac{\tau}{\varphi}\quad \dfrac{s}{k::A}\quad\dfrac{s}{l::A}}{f::B} \qquad \dfrac{\dfrac{\tau}{\varphi}\quad \dfrac{s}{\varnothing::A}}{f::B} \xrightarrow{[w]} \dfrac{\dfrac{\tau}{\varphi}}{f::B}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\tau}{\varphi}\quad\dfrac{\sigma}{\psi}}{f+g::A\wedge B}}{k+l::A\wedge B}\quad\dfrac{\rho}{\theta}}{h::C} \xrightarrow[(\varphi,f\,||\,\psi,g)]{[\wedge]} \dfrac{\dfrac{\dfrac{\dfrac{\tau}{\varphi}}{f::A}}{k::A}\quad\dfrac{\dfrac{\dfrac{\sigma}{\psi}}{g::B}}{l::B}\quad\dfrac{\rho}{\theta}}{h::C}$$

$$\dfrac{\dfrac{\tau}{\varphi}\quad\dfrac{\dfrac{\dfrac{\sigma}{\psi}}{k\triangleright g::A\Rightarrow B}}{f\triangleright l::A\Rightarrow B}\quad\dfrac{\rho}{\theta}}{h::C} \xrightarrow[(\varphi,f\,||\,l,\theta,h)]{[\Rightarrow]} \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\tau}{\varphi}}{f::A}}{k::A}\quad\dfrac{\sigma}{\psi}}{g::B}}{l::B}\quad\dfrac{\rho}{\theta}}{h::C}$$

**Figure 6** Reduction rules.

▶ **Definition 4** (Combinatorial tree). *A **combinatorial tree** $t::C$ with **conclusion** formula $C$ is an inductive tree consisting of either:*

- *a **premiss** $\star::C$, representing (the arena of) $C$, or*
- *a **constant** $c::C$ where $C = P_1\Rightarrow\ldots\Rightarrow P_n\Rightarrow P$ $(n\geq 0)$, or*
- *a **node** $k_1::A_1,\ldots,k_n::A_n\vdash f::C$ with a sequence of **subtrees** $t_1::A_1\ldots t_n::A_n$,*

*written:* $$\dfrac{\dfrac{t_1::A_1}{k_1::A_1}\ \cdots\ \dfrac{t_n::A_n}{k_n::A_n}}{f::C}$$

For a concrete example, Figure 7 gives a reduction featuring various combinatorial trees. We abbreviate $t::C$ to $t$, and write $\tau::\Gamma$ for a ***forest*** $t_1::A_1\ldots t_n::A_n$ (where $\Gamma = A_1,\ldots,A_n$). Edges connecting $\tau$ to antecedents $\varphi = k_1,\ldots,k_n$ are drawn like a single dashed edge, rendering the above tree as $(a)$ below. We indicate a forest of premisses by $\star::\Gamma$, as in $(b)$, and denote the premisses of a tree $t$ by $\star t$. A tree ***for*** the sequent $\Gamma\vdash A$ is one $t::A$ with $\star t = \Gamma$. We visually identify the premisses of a tree by a double dashed edge, as in $(c)$ below for $s$ with $\star s = A,\Delta$. Then $(d)$ is the result of replacing $\star::A$ in $s$ by a tree $t$ for $\Gamma\vdash A$, imitating the cut rule of Figure 5a.

$$(a)\ \dfrac{\dfrac{\tau::\Gamma}{\varphi::\Gamma}}{f::C} \qquad (b)\ \dfrac{\dfrac{\star::\Gamma}{\varphi::\Gamma}}{f::C} \qquad (c)\ \dfrac{\star::A\quad\star::\Delta}{s::B} \qquad (d)\ \dfrac{\dfrac{\star::\Gamma}{t::A}\quad\star::\Delta}{s::B}$$

▶ **Definition 5** (Reduction). *Reduction of combinatorial trees is by the rules in Figure 6.*

The reduction rules are essentially those of the sequent calculus, but in a setting that is free of permutations. Observe that while combinatorial trees involve a good amount of notation, the notion of a tree of normal forms is in fact highly conceptual. For reduction, the particular use of ICPs is secondary, and any representation of normal forms would do: the reduction rules are determined entirely by the *sequentialization* or *decomposition* of nodes.

We will assume that constants represent primitives of base type, such as integers and booleans, and functions over base types, such as addition. We extend the reduction rule $[\Rightarrow]$ to the latter case as below; an example instance would be where $c$ is the integer 7 and $c'$ is a squaring function, with the resulting constant $c''$ the integer 49.

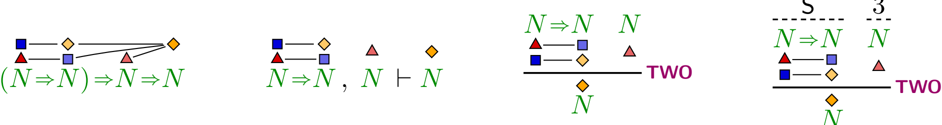
$$\dfrac{\dfrac{c}{1::P}\quad\dfrac{c'}{1\triangleright k::P\Rightarrow A}\quad\dfrac{\tau}{\varphi}}{f::B} \xrightarrow[(1,1\ ||\ k,\varphi,f)]{[\Rightarrow]} \dfrac{\dfrac{c''}{k::A}\quad\dfrac{\tau}{\varphi}}{f::B}$$

### 3.1 Reduction examples

We illustrate reduction with an example analogous to the following lambda calculus reduction, applying the Church numeral two $\lambda f.\lambda x.f(fx) : (N \Rightarrow N) \Rightarrow N \Rightarrow N$ to the squaring function constant $\mathsf{S} : N \Rightarrow N$ and the integer constant $3 : N$.

$$(\lambda f.\lambda x.f(fx))\,\mathsf{S}\,3 \;\rightarrow\; (\lambda x.\mathsf{S}(\mathsf{S}x))\,3 \;\rightarrow\; \mathsf{S}(\mathsf{S}3) \;\rightarrow\; \mathsf{S}9 \;\rightarrow\; 81$$

The combinatorial proof **TWO** corresponding to the Church numeral is the penultimate one displayed in Figure 3. Below, from left to right, we have: numeral two in compact form; two in sequent form; two as a node in a combinatorial tree; and the combinatorial tree representing $(\lambda f.\lambda x.f(fx))\,\mathsf{S}3$.




The reduction sequence is as follows:

S 3 N⇒N N TWO N $\xrightarrow{[c]}$ S S 3 N⇒N N⇒N N N $\xrightarrow{[\Rightarrow]}$ S 9 N⇒N N N $\xrightarrow{[\Rightarrow]}$ 81 N N $\xrightarrow{[1]}$ 81

For a richer example we consider the ICP version of the Church successor $\lambda n.\lambda f.\lambda x.f(nfx)$ applied to Church zero $\lambda f.\lambda x.x$, the squaring function $\mathsf{S} : N \Rightarrow N$ and 4, to yield 16.

$$(\lambda n.\lambda f.\lambda x.f(nfx))\;\;(\lambda f.\lambda x.x)\;\mathsf{S}\;4 \qquad \twoheadrightarrow \qquad 16$$

The ICP reduction is shown in Figure 7.

## 4 Strong Reduction

The reduction rules $[\wedge], [\Rightarrow]$ apply only when the two ports involved are both open (this is what the side-conditions on the reduction rules entail). We briefly show that this does not lead to a deadlock. In a combinatorial tree, a port is ***extremal*** if it is connected to a premiss or the consequent of the root node, otherwise ***internal***.

▶ **Lemma 6** (Progress). *For a combinatorial tree $t$ with at least one edge, if no extremal port is open, then a reduction step applies.*

The progress lemma illustrates a limitation of the normalization process: reduction may become deadlocked if an extremal port remains open. This is closely related to *weak* reduction in the $\lambda$-calculus, which does not reduce under an abstraction, though note it is not the same: internal reduction in a combinatorial tree is allowed, and may still be possible, when the root node is an abstraction. As with weak reduction, this is no limitation in practice: we expect a real program to be of base type, and without free variables (the premisses of a combinatorial tree). In that case the progress lemma guarantees we will not reach a deadlock. This explains also the reason to include constants: without them it is impossible to create a combinatorial tree of base type with no premisses, as it would be logically unsound.

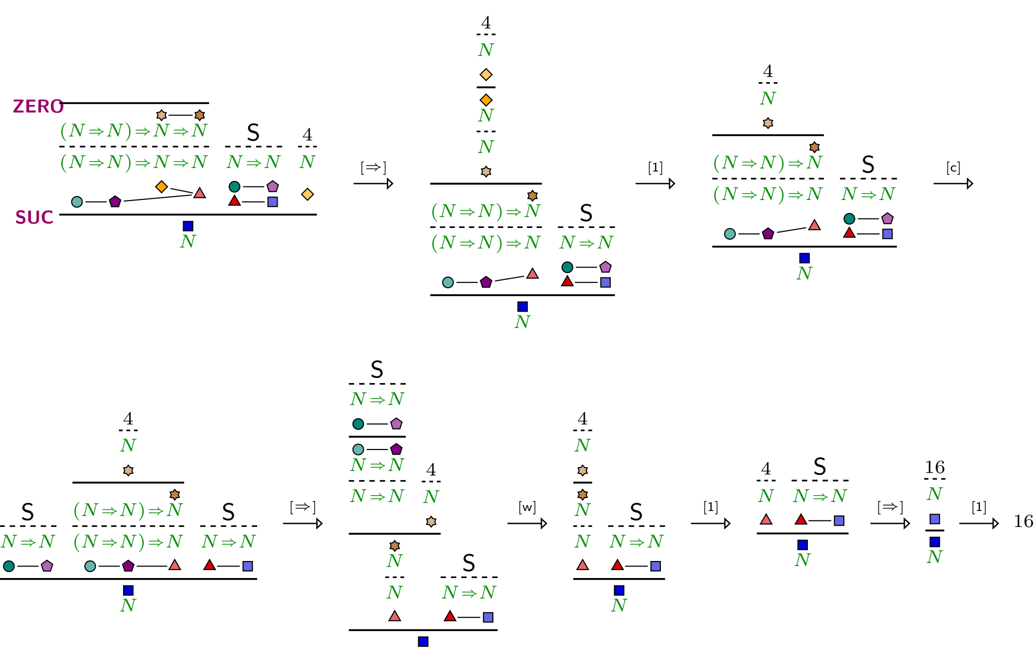



**Figure 7** Example of ICP normalization corresponding to the lambda calculus normalization of the Church successor function applied to Church zero, the squaring function constant S, and the constant 4: $(\lambda n.\lambda f.\lambda x.f(nfx))\ (\lambda f.\lambda x.x)$ S $4\ \to^* 16$.

To reduce any combinatorial tree, we combine reduction with sequentialization. We may then reduce open extremal ports by interpreting them as sequent rules. We add a special axiom (icp), given below, to the cut-free sequent calculus. It incorporates a combinatorial tree $t$ for $\Gamma \vdash A$ as a sub-proof of $\Gamma \vdash A$. A proof in this calculus is a ***hybrid proof***.

$$\frac{\boxed{t :: A}}{\star t \vdash A}\ (\mathsf{icp})$$

The reduction rules [1], [∧], and [⇒] apply directly to hybrid proofs, since they preserve the premisses and conclusion of a combinatorial tree. The rules [c] and [w] duplicate or delete premisses; to accommodate this in hybrid proofs, contraction or weakening rules are added. The resulting rules are the last two in Figure 8, which gives the rules for strong reduction.

▶ **Definition 7** (Hybrid reduction). ***Hybrid proof reduction*** *is the rewrite relation on hybrid proofs generated by the rules* [1], [∧], [⇒] *in Figure 6 plus the rules in Figure 8.*

Progress (Lemma 6) gives the following.

▶ **Lemma 8** (Hybrid progress). *If a hybrid proof contains an* (icp) *axiom, a hybrid reduction step applies.*

A normal form of a hybrid proof is then a regular, cut-free sequent proof. This may directly be used to construct an ICP, to obtain fully general ICP normalization. The effect of embedding a combinatorial tree in a hybrid proof is akin to *normalization-by-evaluation* [5]: it provides an environment that supplies sufficient arguments to any function (it is an *applicative context*), and other similar services, to ensure continued reduction.

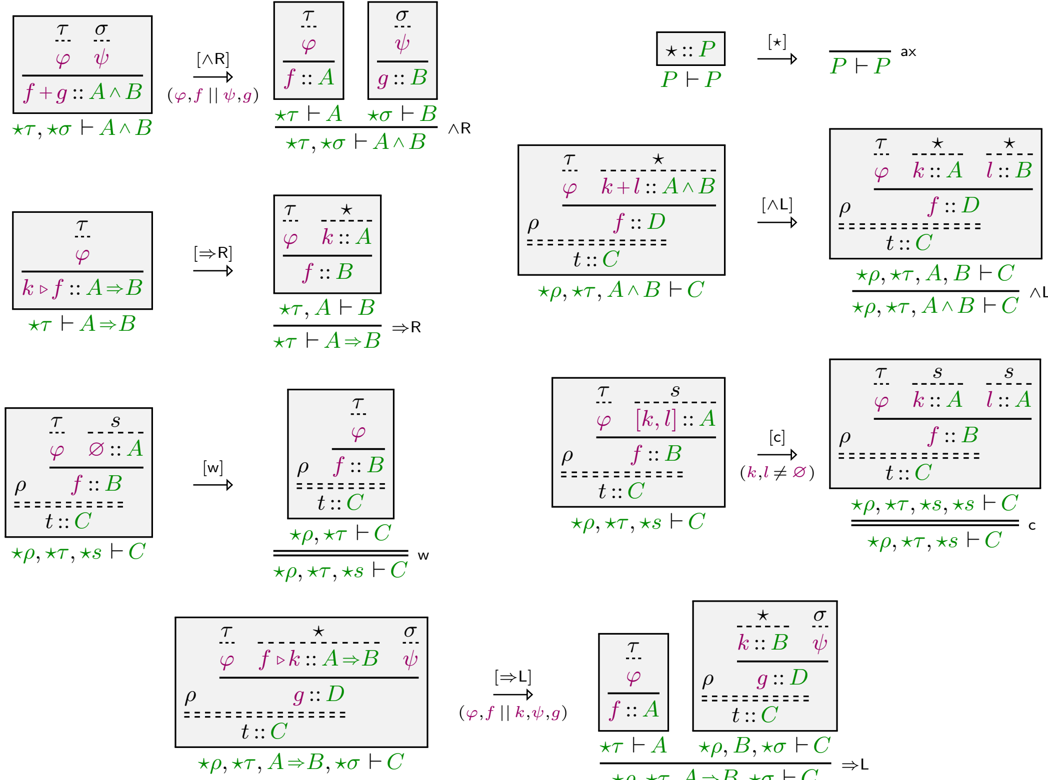


**Figure 8** Hybrid sequentialization and reduction rules.

## 5 Confluence and strong normalization

Combinatorial-tree reduction is confluent and strongly normalizing. In this section we will consider only *local confluence*, which demonstrates the intricacies arising from the local canonicity property of ICPs. Confluence then follows from strong normalization by Newman's Lemma.

The reduction rules for ICPs interact in several intricate ways. Not only can a single node have multiple redexes along different edges, even a single edge may reduce in more than one way. This is due to the multiple ways an arena morphism can be composed inductively, which factor out the formula isomorphisms of associativity, symmetry, and currying, as well as the interaction of conjunction with contraction. Concretely, we have the following equations:

$$f+g = g+f \qquad\qquad \varnothing+\varnothing = \varnothing$$
$$f+(g+h) = (f+g)+h \qquad\qquad [k,\varnothing] = k$$
$$(k+l) \triangleright f = k \triangleright (l \triangleright f) \qquad\qquad [k_1,k_2]+[l_1,l_2] = [k_1+l_1, k_2+l_2]$$

We recognize two kinds of critical pairs:

**Single-edge** when multiple reduction steps apply to a single cut-edge, due to the above equations;

**Single-node** when multiple reduction steps on distinct edges split the same node.

We do not consider non-splitting reductions on different edges of the same node as critical pairs, since the reductions are independent and converge immediately.

Figure 9 shows how the critical pairs converge. In the following, we will explain the notation used, and consider the precise equations that give rise to the single-edge diagrams.

We use $\twoheadrightarrow$ for the reflexive-transitive closure of $\rightharpoonup$, and dashed arrows are implied by the diagram. Note that the last four diagrams use a different colour scheme to help identify arena morphisms and subtrees across reduction steps.

The first five diagrams cover the single-edge critical pairs, and the last three the single-node critical pairs. The latter, $[\Rightarrow]/[\Rightarrow]^2$, $[\wedge]/[\Rightarrow]$, and $[\Rightarrow]/[\Rightarrow]^3$, are similar to critical pairs found in $\lambda$-calculi and proof nets, and converge accordingly.

The single-edge critical pairs are new and delicate. We introduce the notation $t+s$ to mean the following.

$$t+s = \frac{\overset{\tau}{\varphi} \quad \overset{\sigma}{\psi}}{f+g} \qquad \text{where} \qquad t = \frac{\overset{\tau}{\varphi}}{f} \qquad s = \frac{\overset{\sigma}{\psi}}{g}$$

In the first four diagrams in Figure 9, we depict only the ports and subtrees involved, but omit the node they are attached to. The five single-edge confluence diagrams are due to the following equations:

$$\begin{array}{lrcl}
[\wedge]/[\mathsf{w}]: & \varnothing+\varnothing & = & \varnothing \\
[\wedge]/[\wedge]: & k+(l+m) & = & (k+l)+m \\
[\Rightarrow]/[\Rightarrow]^1: & (k+l)\triangleright f & = & k\triangleright(l\triangleright f)
\end{array}
\qquad
\begin{array}{lrcl}
[\wedge]/[\mathsf{c}]^1: & [k_1,k_2]+[l_1,l_2] & = & [k_1+l_1,k_2+l_2] \\
[\wedge]/[\mathsf{c}]^2: & [k_1,k_2]+l & = & [k_1+l,k_2+\varnothing]
\end{array}$$

Since the eight diagrams in Figure 9 cover all cases of single-edge and single-node critical pairs, we have the following proposition.

▶ **Proposition 9.** *Reduction $\rightharpoonup$ is locally confluent.*

The strong normalization property is stated without proof; the proofs can be found in the appendix of the technical report on HAL [17].

▶ **Theorem 10** (Strong normalization). *Combinatorial-tree reduction is strongly normalizing.*

## 6 Combinatory lambda-calculus

To further illustrate the reduction process, we connect ICPs to the $\lambda$-calculus, via an explicit-substitution $\lambda$-calculus that we call the ***combinatory $\lambda$-calculus***. The calculus is a Curry–Howard interpretation of sequent calculus, of the kind studied by Graham-Lengrand [28]. We include constants $c$ to match those of combinatorial trees.

▶ **Definition 11.** *The **combinatory $\lambda$-calculus** has **normal terms** $N, M$, **patterns** $p, q$, and **terms** $S, T$ given by the following grammars.*

$$M, N ::= x \mid \langle M, N\rangle \mid \lambda p.\, M \mid M[p \leftarrow\!\!\shortmid xN]$$

$$p, q ::= x \mid \langle p, q\rangle \qquad S, T ::= c \mid M[p_1 \leftarrow T_1, \ldots, p_n \leftarrow T_n]$$

*The **binding variables** $\mathsf{bv}(p)$ of $p$ and the **free variables** $\mathsf{fv}(M)$ of $M$ are as follows; in $M[p \leftarrow\!\!\shortmid xN]$ we require that $\mathsf{fv}(M) \cap \mathsf{bv}(p) \neq \varnothing$, and in $\langle p, q\rangle$ that $\mathsf{bv}(p) \cap \mathsf{bv}(q) = \varnothing$.*

$$\begin{array}{rcl}
\mathsf{bv}(x) & = & x \\
\mathsf{fv}(x) & = & x \\
\mathsf{fv}(\lambda p.\, M) & = & \mathsf{fv}(M) - \mathsf{bv}(p)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{bv}(\langle p, q\rangle) & = & \mathsf{bv}(p) \cup \mathsf{bv}(q) \\
\mathsf{fv}(\langle M, N\rangle) & = & \mathsf{fv}(M) \cup \mathsf{fv}(N) \\
\mathsf{fv}(M[p \leftarrow\!\!\shortmid xN]) & = & (\mathsf{fv}(M) - \mathsf{bv}(p)) \cup \{x\} \cup \mathsf{fv}(N)
\end{array}$$

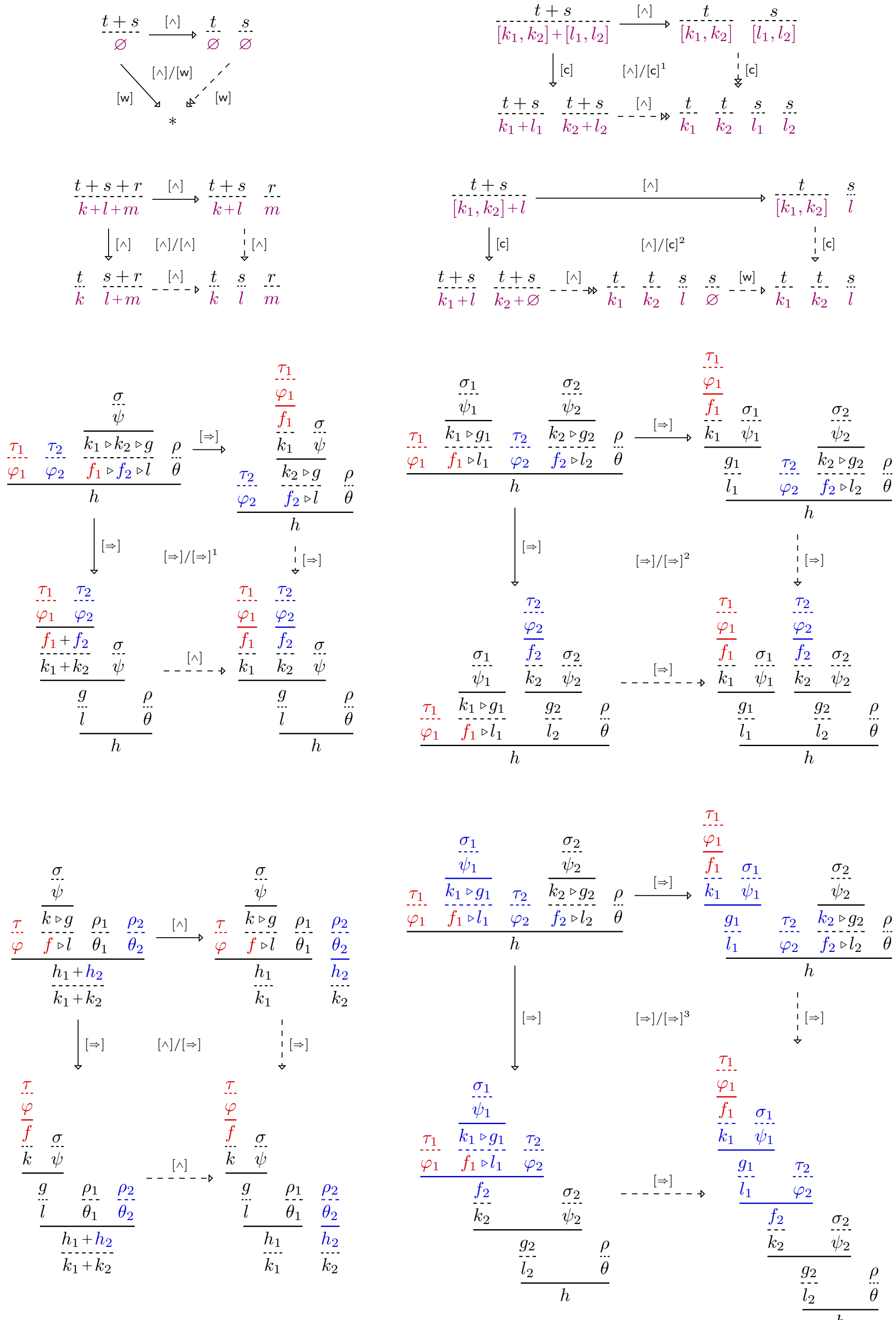


**Figure 9** Single-edge and single-node confluence diagrams.

$$\frac{}{1 \vdash 1 \;\mapsto\; x:P \vdash x:P}\langle\!\langle \mathsf{ax} \rangle\!\rangle \qquad \frac{\varphi, k, l \vdash f \;\mapsto\; \Gamma, p:A, q:B \vdash M:C}{\varphi, k+l \vdash f \;\mapsto\; \Gamma, \langle p,q\rangle : A \wedge B \vdash M:C}\langle\!\langle \wedge\mathsf{L} \rangle\!\rangle$$

$$\frac{\varphi \vdash f \;\mapsto\; \Gamma \vdash M:C}{\varphi, \varnothing \vdash f \;\mapsto\; \Gamma, p:A \vdash M:C}\langle\!\langle \mathsf{w} \rangle\!\rangle \qquad \frac{\varphi \vdash f \;\mapsto\; \Gamma \vdash M:A \qquad \psi \vdash g \;\mapsto\; \Delta \vdash N:B}{\varphi, \psi \vdash f+g \;\mapsto\; \Gamma, \Delta \vdash \langle M,N\rangle : A \wedge B}\langle\!\langle \wedge\mathsf{R} \rangle\!\rangle$$

$$\frac{\varphi, k, l \vdash f \;\mapsto\; \Gamma, p:A, p:A \vdash M:C}{\varphi, [k,l] \vdash f \;\mapsto\; \Gamma, p:A \vdash M:C}\langle\!\langle \mathsf{c} \rangle\!\rangle \qquad \frac{\varphi \vdash f \;\mapsto\; \Gamma \vdash N:A \qquad k, \psi \vdash g \;\mapsto\; p:B, \Delta \vdash M:C}{\varphi, f \triangleright k, \psi \vdash g \;\mapsto\; \Gamma, x:A \Rightarrow B, \Delta \vdash M[p \leftarrow\!\!\!\!\leftarrow xN]:C}\langle\!\langle \Rightarrow\mathsf{L} \rangle\!\rangle$$

$$\frac{\varphi, k \vdash f \;\mapsto\; \Gamma, p:A \vdash M:B}{\varphi \vdash k \triangleright f \;\mapsto\; \Gamma \vdash \lambda p.\, M : A \Rightarrow B}\langle\!\langle \Rightarrow\mathsf{R} \rangle\!\rangle$$

**Figure 10** From ICPs to simply-typed combinatory $\lambda$-terms.

In $\lambda p.\, M$, $M[p \leftarrow\!\!\!\!\leftarrow xN]$, and $M[p_1 \leftarrow T_1, \dots, p_n \leftarrow T_n]$ the variables in the patterns $p$ and $p_i$ bind in $M$. The construction $M[p \leftarrow\!\!\!\!\leftarrow xN]$ is a ***shared application***, with a variable $x$ as function and the term $N$ as argument, where the pattern $p$ may bind variables with multiple occurrences in $M$. The condition that $\mathsf{bv}(p)$ and $\mathsf{fv}(M)$ must intersect means at least one variable becomes bound; this corresponds to the condition (‡) on the rule $\Rightarrow$L for ICPs in Figure 2 (that the consequent of a left-implication must not be introduced by weakening). The construction $[p_1 \leftarrow T_1, \dots, p_n \leftarrow T_n]$ is an ***environment***, and corresponds to attaching the subtrees to a node in a combinatorial tree. We abbreviate it by $[e]$, or $[p_1 \leftarrow T_1, e]$, etc.

▶ **Definition 12.** *Figure 10 gives the (non-deterministic)* ***translation*** *from ICPs to simply-typed, normal terms of the combinatory $\lambda$-calculus. We extend it to combinatorial trees as follows: $\mapsto$ is the identity on constants, and if*

$$k_1, \dots, k_n, \varphi \vdash f \qquad \mapsto \qquad p_1 : A_1, \dots, p_n : A_n, \Delta \vdash M : B$$

*and if $t_i \;\mapsto\; \Gamma_i \vdash T_i : A_i$ (with $t_i \neq \star$) for all $i \leq n$, then*

$$\frac{\overset{t_1}{k_1} \;\cdots\; \overset{t_n}{k_n} \;\; \overset{\star}{\varphi}}{f} \qquad \mapsto \qquad \Gamma_1, \dots, \Gamma_n, \Delta \vdash M[p_1 \leftarrow T_1, \dots, p_n \leftarrow T_n] : B\ .$$

The shared applications $[p \leftarrow\!\!\!\!\leftarrow xN]$ of the combinatory $\lambda$-calculus are subject to permutations, creating an equivalence $\sim$ on terms. We define it below, where we abbreviate $[p \leftarrow\!\!\!\!\leftarrow xN]$ by $[a]$, with $\mathsf{bv}(a) = \mathsf{bv}(p)$ and $\mathsf{fv}(a) = \{x\} \cup \mathsf{fv}(M)$.

$$\begin{array}{rcll}
\langle M[a], N\rangle &\sim& \langle M,N\rangle[a] & \mathsf{bv}(a) \cap \mathsf{fv}(N) = \varnothing \\
\langle M, N[a]\rangle &\sim& \langle M,N\rangle[a] & \mathsf{bv}(a) \cap \mathsf{fv}(M) = \varnothing \\
\lambda p.\,(M[a]) &\sim& (\lambda p.\, M)[a] & \mathsf{bv}(p) \cap \mathsf{fv}(a) = \varnothing \\
M[p \leftarrow\!\!\!\!\leftarrow xN[a]] &\sim& M[p \leftarrow\!\!\!\!\leftarrow xN][a] & \mathsf{bv}(a) \cap \mathsf{fv}(N) = \varnothing \\
M[a][b] &\sim& M[b][a] & \mathsf{bv}(b) \cap \mathsf{fv}(a) = \varnothing, \mathsf{bv}(a) \cap \mathsf{fv}(b) = \varnothing
\end{array}$$

The above equivalence factors out sequent calculus permutations. We will further assume combinatory $\lambda$-terms equivalent modulo the formula-isomorphisms (symmetry, associativity, and currying). These are factored out simply by considering patterns modulo these rules, but there is a catch: patterns and pairs are connected through cuts, or explicit substitutions, and laws must be applied to both simultaneously. We show an example with currying to demonstrate that a full definition is intricate, and leave it implicit.

$$M[z \leftarrow\!\!\!\!\leftarrow x\langle P,Q\rangle][x \leftarrow \lambda\langle p,q\rangle.\, N] \quad\sim\quad M[z \leftarrow\!\!\!\!\leftarrow yQ][y \leftarrow\!\!\!\!\leftarrow xP][x \leftarrow \lambda p.\lambda q.\, N]$$

With the above equivalence on terms, the following is a direct corollary of local canonicity (Theorem 3).

▶ **Proposition 13.** *Combinatorial trees canonically represent typed combinatory λ-terms:*

$$S \sim T \quad \Longleftrightarrow \quad \exists t.\ t \Mapsto S \ \wedge\ t \Mapsto T$$

We reduce combinatory λ-terms modulo the equivalence $\sim$. We write $\{T/x\}$ for the substitution of $x$ by $T$, and if the patterns $p, q$ are isomorphic as trees and $\mathsf{bv}(p) \cap \mathsf{bv}(q) = \varnothing$ then $\{q/p\}$ is the substitution induced by

$$\{\langle q_1, q_2\rangle / \langle p_1, p_2\rangle\} = \{q_1/p_1\}\{q_2/p_2\} \ .$$

▶ **Definition 14.** *Reduction of combinatory λ-terms modulo $\sim$ is by the following rules, where: $[e_P]$ and $[e_Q]$ bind only in $P$ respectively $Q$; in $\langle\Rightarrow\rangle$ we require $x \notin \mathsf{fv}(P) \cup \mathsf{fv}(Q)$; in $\langle\mathsf{c}\rangle$ we require $\mathsf{bv}(q) \cap \mathsf{fv}(M) \neq \varnothing$; and in $\langle\mathsf{w}\rangle$ that $\mathsf{bv}(p) \cap \mathsf{fv}(M) = \varnothing$.*

$$\begin{aligned}
M[x \leftarrow y[e], e'] &\xrightarrow{\langle 1\rangle} M\{y/x\}[e, e'] \\
M[\langle p, q\rangle \leftarrow \langle P, Q\rangle[e_P, e_Q], e] &\xrightarrow{\langle\wedge\rangle} M[p \leftarrow P[e_P], q \leftarrow Q[e_Q], e] \\
P[p \leftarrow xQ][e_Q, x \leftarrow \lambda q.\, M[e], e_P] &\xrightarrow{\langle\Rightarrow\rangle} P[p \leftarrow M[q \leftarrow Q[e_Q], e], e_P] \\
M\{p/q\}[p \leftarrow T, e] &\xrightarrow{\langle\mathsf{c}\rangle} M[q \leftarrow T, p \leftarrow T, e] \\
M[p \leftarrow T, e] &\xrightarrow{\langle\mathsf{w}\rangle} M[e]
\end{aligned}$$

Comparing the reduction rules with the corresponding ones for ICPs in Figure 6, together with Proposition 13, gives:

▶ **Proposition 15.** *Reduction on ICPs and combinatory λ-terms (modulo equivalence) commutes with interpretation*

$$\begin{array}{ccc}
t & \xrightarrow{[x]} & s \\
\Downarrow & & \Downarrow \\
T & \xrightarrow{\langle x\rangle} & S
\end{array}$$

The comparison with λ-calculus allows us to make a further observation. ICP normalization is a form of *closed reduction* [7] (there called *weak reduction*), where a redex $(\lambda x.M)N$ may not be reduced if $N$ contains free variables that are bound by the surrounding context. This has the benefit to implementation that alpha-conversion becomes unnecessary. Our construction of combinatorial trees is even stronger: it is impossible to construct such a redex, or to produce one by reduction. This can be observed from the combinatory λ-calculus, which does not support abstraction at the level of terms $T$, only at the level of normal terms.

Abstraction on terms can be introduced as a defined operation, called ***lambda-lifting*** [26]. The analogous operation on ICP combinatorial trees would be a transformation

$$\frac{\star :: A \quad \star :: \Gamma}{t :: B} \quad \mapsto \quad \frac{\star :: \Gamma}{t' :: A \Rightarrow B} \ .$$

We can perform it by abstracting over $\star :: A$ locally, in the node where it resides, and transform every node on the path from there to the root as follows,

$$\frac{k :: C \quad \varphi}{f :: D} \quad \mapsto \quad \frac{i \triangleright k :: A \Rightarrow C \quad \varphi}{i \triangleright f :: A \Rightarrow D}$$

where the port $k :: C$ is that on the path to $\star :: A$, and the arena morphism $i : [\![A]\!] \to [\![A]\!]$ is the identity on $[\![A]\!]$. In effect, one is threading the abstraction over $A$ *through* the cuts in the tree, rather than adding it as a connection *outside* of them.

By way of example, below is the reduction corresponding to the ICP normalization sequence in Figure 7.

$$\begin{array}{rl|rl}
 & v[v \leftarrowtail gw][w \leftarrowtail yz][y \leftarrowtail ng][n \leftarrow \lambda f.\lambda x.x, g \leftarrow S, z \leftarrow 4] & & \dots \\
\sim & v[v \leftarrowtail gw][w \leftarrowtail yg][y \leftarrowtail nz][n \leftarrow \lambda x.\lambda f.x, z \leftarrow 4, g \leftarrow S] & \xrightarrow{\langle \mathsf{w} \rangle} & v[v \leftarrowtail gw][w \leftarrow x[x \leftarrow 4], g \leftarrow S] \\
\xrightarrow{\langle \Rightarrow \rangle} & v[v \leftarrowtail gw][w \leftarrowtail yg][y \leftarrow \lambda f.x[x \leftarrow z[z \leftarrow 4]], g \leftarrow S] & \xrightarrow{\langle 1 \rangle} & v[v \leftarrowtail gw][w \leftarrow 4, g \leftarrow S] \\
\xrightarrow{\langle 1 \rangle} & v[v \leftarrowtail gw][w \leftarrowtail yg][y \leftarrow \lambda f.x[x \leftarrow 4]], g \leftarrow S] & \xrightarrow{\langle \Rightarrow \rangle} & v[v \leftarrow 16] \\
\xrightarrow{\langle \mathsf{c} \rangle} & v[v \leftarrowtail gw][w \leftarrowtail yh][y \leftarrow \lambda f.x[x \leftarrow 4]], g \leftarrow S, h \leftarrow S] & \xrightarrow{\langle 1 \rangle} & 16 \\
\xrightarrow{\langle \Rightarrow \rangle} & v[v \leftarrowtail gw][w \leftarrow x[f \leftarrow h[h \leftarrow S], x \leftarrow 4], g \leftarrow S] \dots & &
\end{array}$$

## 7 Supercombinators

Supercombinators [24] are the basis of an efficient implementation of functional programming [29]. The main reason for their efficiency is that expressions are compiled into trees (or graphs) over a fixed set of operators, each given as an instruction set that implements the appropriate reduction sequence.

▶ **Definition 16.** ***Supercombinators*** *$C, D$ and* ***supercombinator expressions*** *$E_X$, $F_X$, where $X$ is a set of variables, are given by the following grammars.*

$$C, D ::= \lambda x_1 \dots \lambda x_n . E_{\{x_1,\dots,x_n\}} \qquad E_X, F_X ::= x \in X \mid C \mid F_X\, E_X$$

The set $X$ restricts which variables may occur free in a supercombinator expression, so that each supercombinator is a closed term; we may omit it as superscript for brevity. The grammar for supercombinators $C$ may be extended to include constants. Reduction is *weak head reduction* on an expression $E_\varnothing$, as given by the rule below. It applies only at top-level, not in context, and if there are fewer than $n$ arguments to a supercombinator with $n$ abstractions, reduction halts.

$$(\lambda x_1 \dots \lambda x_n . E)\, F_1 \dots F_n\, F_{n+1} \dots F_{n+m} \;\mapsto\; E\{F_1/x_1\} \dots \{F_n/x_n\}\, F_{n+1} \dots F_{n+m}$$

During reduction, substitutions are applied only to the top-level $E_\varnothing$ expression, and not to supercombinators, which remain fixed. This allows them to be compiled into instruction sets to carry out the appropriate reduction by the rule $\mapsto$ above.

Structurally, supercombinators are trees or graphs where each node is a supercombinator $C$ in which each occurring supercombinator $D$ is considered as a *pointer* to the node for $D$. This is highly similar to combinatorial trees, which feature the same tree structure except with ICPs for nodes. The main dissimilarities between supercombinators and combinatorial trees are then as follows.

- Supercombinator reduction is by an abstract machine, where combinatorial-tree reduction is a variant of cut-elimination.
- Supercombinators are trees over $\beta$-normal $\lambda$-terms where abstractions may not occur under an application, where nodes in combinatorial trees are $\eta$-expanded $\beta$-normal sequent proofs modulo permutations.

These differences are conceptually shallow, but risk burying a formal comparison in technicalities. We will therefore interpret supercombinators in the combinatory $\lambda$-calculus instead (which, mainly, does not require $\eta$-expansion), and simulate reduction only up to explicit substitutions.

▶ **Definition 17.** *The relations $\blacktriangleright$ and $\triangleright$, defined inductively below, interpret supercombinators respectively supercombinator expressions into the combinatory $\lambda$-calculus.*

$$\frac{E \triangleright M[e]}{\lambda x_1 \dots x_n.E \blacktriangleright (\lambda x_1 \dots x_n.M)[e]} \qquad \frac{}{x \triangleright x} \qquad \frac{C \blacktriangleright T}{C \triangleright x[x \leftarrow T]} \qquad \frac{E \triangleright x[a_1]\dots[a_k][e] \qquad F \triangleright M[f]}{EF \triangleright y[y \hookleftarrow xM][a_1]\dots[a_k][e,f]}$$

Note how this indeed translates a supercombinator to a term $(\lambda x_1 \dots \lambda x_n.N)[e]$ consisting of a normal form $\lambda x_1 \dots \lambda x_n.N$ with a subtree for each occurring supercombinator in the explicit substitutions $[e]$. To simulate reduction, a reduct is translated as follows.

$$\dfrac{\dfrac{\dfrac{E \triangleright M[e]}{\lambda x_1 \dots \lambda x_n.E \blacktriangleright (\lambda x_1 \dots \lambda x_n.M)[e]}}{\lambda x_1 \dots \lambda x_n.E \triangleright y[y \leftarrow (\lambda x_1 \dots \lambda x_n.M)[e]]} \qquad F_1 \triangleright N_1[f_1] \quad \dots \quad F_n \triangleright N_n[f_n]}{(\lambda x_1 \dots \lambda x_n.E)\, F_1 \dots F_n \triangleright z_n[z_n \hookleftarrow z_{n-1}N_n]\dots[z_1 \hookleftarrow yN_1][y \leftarrow (\lambda x_1 \dots \lambda x_n.M)[e], f_1, \dots, f_n]}$$

Reduction for this term proceeds as follows.

$$\begin{aligned} & z_n[z_n \hookleftarrow z_{n-1}N_n]\dots[z_2 \hookleftarrow z_1 N_2][z_1 \hookleftarrow yN_1][y \leftarrow (\lambda x_1.\lambda x_2 \dots \lambda x_n.M)[e], f_1, f_2, \dots, f_n] \\ \xrightarrow{\langle\Rightarrow\rangle}\; & z_n[z_n \hookleftarrow z_{n-1}N_n]\dots[z_2 \hookleftarrow z_1 N_2][z_1 \leftarrow (\lambda x_2 \dots \lambda x_n.M)[x_1 \leftarrow N_1[f_1], e], f_2, \dots, f_n] \\ \xrightarrow{\langle\Rightarrow\rangle}\!\!\!\rightarrow\; & z_n[z_n \leftarrow M[x_1 \leftarrow N_1[f_1], \dots, x_n \leftarrow N_n[f_n], e]] \end{aligned}$$

The result corresponds to the supercombinator reduct $E\{F_1/x_1\}\dots\{F_n/x_n\}$, except that the explicit substitutions $[x_i \leftarrow N_i[f_i]]$ are not evaluated as substitutions. They cannot be: combinatory $\lambda$-term reduction does not differentiate between the interpretation of the top-level supercombinator expression $E_\varnothing$ on which reduction takes place, and which does admit substitutions, and internal subcombinator expressions which do not. We will therefore contend ourselves with the "moral" equivalence of both reductions.

# 8 Lambda-calculus

To complete the exposition, we map the combinatory $\lambda$-calculus onto the regular $\lambda$-calculus with pairing. We have the following terms and rewrite rules, where $i \in \{1,2\}$.

$$M,N ::= x \mid \lambda x.M \mid MN \mid \pi_i M \mid \langle M,N\rangle \qquad (\lambda x.M)N \rightarrow_\beta M\{N/x\} \qquad \pi_i\langle M_1,M_2\rangle \rightarrow_\pi M_i$$

The translation from combinatory $\lambda$-terms into $\lambda$-terms $\lfloor\cdot\rfloor$ is as follows, where we substitute for a pattern via $\{M/\langle p,q\rangle\} = \{\pi_1 M/p, \pi_2 M/q\}$.

$$\begin{aligned} \lfloor x \rfloor &= x \\ \lfloor \langle M,N\rangle \rfloor &= \langle \lfloor M\rfloor, \lfloor N \rfloor\rangle \\ \lfloor \lambda p.M \rfloor &= \lambda x.\lfloor M\rfloor\{x/p\} \\ \lfloor M[p \hookleftarrow xN] \rfloor &= \lfloor M \rfloor\{x\lfloor N\rfloor/p\} \\ \lfloor M[p_1 \leftarrow T_1, \dots, p_n \leftarrow T_n] \rfloor &= \lfloor M\rfloor\{\lfloor T_1\rfloor/p_1\}\dots\{\lfloor T_n\rfloor/p_n\} \end{aligned}$$

The combined translation then takes ICP combinatorial trees to $\lambda$-terms. As with the combinatory $\lambda$-calculus, we assume $\lambda$-terms equivalent ($\sim$) modulo formula-isomorphisms (symmetry, associativity, currying). Sequent permutations are already naturally factored out, but at the cost of exponential growth. We will demonstrate this here.

In the combinatory λ-calculus, the reason that an application must occur in an explicit substitution is precisely that the consequent of a left-implication may have been contracted, the situation highlighted in the introduction:

$$\dfrac{\Gamma\vdash A \qquad \dfrac{B,B,\Delta\vdash C}{B,\Delta\vdash C}\,\mathsf{c}}{\Gamma,A\Rightarrow B,\Delta\vdash C}\,{\Rightarrow}\mathsf{L} \qquad\approx\qquad \dfrac{\dfrac{\Gamma\vdash A \qquad \dfrac{\Gamma\vdash A \quad B,B,\Delta\vdash C}{B,\Gamma,A\Rightarrow B,\Delta\vdash C}\,{\Rightarrow}\mathsf{L}}{\Gamma,A\Rightarrow B,\Gamma,A\Rightarrow B,\Delta\vdash C}\,{\Rightarrow}\mathsf{L}}{\Gamma,A\Rightarrow B,\Delta\vdash C}\,\mathsf{c}$$

The corresponding equivalence on combinatory terms is:

$$M\{p/q\}[p \leftarrow xN] \qquad \approx \qquad M[q \leftarrow xN][p \leftarrow xN]$$

(where $\mathsf{bv}(q)\cap\mathsf{fv}(M) \neq \varnothing$), while both translate to the same λ-term $\lfloor M\rfloor\{x\lfloor N\rfloor/p\}$. Repeated duplication incurred in this way gives rise to exponential growth.

Let ***strong equivalence*** $S \approx T$ on combinatory λ-terms be the equivalence generated by the above and $\sim$. We have the following proposition.

▶ **Proposition 18.** *For combinatory λ-terms $S, T$, we have*

$$S \approx T \iff \lfloor S\rfloor = \lfloor T\rfloor \ .$$

## 9 Conclusion

We have given a direct and natural account of normalization for intuitionistic combinatorial proofs. We believe our approach of *external rewriting*, here manifested in the notion of *combinatorial tree*, applies much more broadly, in the following two ways.

Firstly, specifically for the present, intuitionistic case, our notion of composition is highly abstract: what we have are simply trees of normal forms, with the natural reduction rules given by the meta-level sequent calculus. As a generalization of super-combinators, a correspondence we aim to make more precise in future work, we hope that our approach leads to improvements in compiler design. Perhaps the ability to express all normal forms, and the more fine-grained reduction steps, will allow more efficient program transformations, while retaining the benefits of super-combinators.

Secondly, our aim has been towards a notion of composition for combinatorial proofs in general, and to illustrate this we briefly sketch how our construction applies to classical combinatorial proofs [18]. Our combinatorial trees generalize to *combinatorial graphs*, which are still connected and acyclic (i.e. still a mathematical tree), but without a designated root. Nodes are classical combinatorial proofs over one-sided sequents, and edges are cuts connecting dual formulae. As may be expected of a semantic account of classical cut-elimination, one does not obtain strong normalization because of the Lafont examples [14] (specifically, a cut on two contracted formulae), but weak normalization is expected to hold. This is the subject of current work.

## References


1 Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 1996.

2 Matteo Acclavio and Lutz Straßburger. On combinatorial proofs for logics of relevance and entailment. In Rosalie Iemhoff and Michael Moortgat, editors, *26th Workshop on Logic, Language, Information and Computation (WoLLIC 2019)*. Springer, 2019.

3 Matteo Acclavio and Lutz Straßburger. On combinatorial proofs for modal logic. In Serenella Cerrito and Andrei Popescu, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings*, volume 11714 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2019. doi:10.1007/978-3-030-29026-9_13.

4 Gianluigi Bellin and Jacques van de Wiele. Subnets of proof-nets in MLL$^{-}$. In *Advances in Linear Logic*, pages 249–270, 1995.

5 Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *6th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–212, 1991.

6 Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

7 Naim Çağman and J. Roger Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1–2):239–249, 1998.

8 Roberto Di Cosmo. A short survey of isomorphisms of types. *Mathematical structures in computer science*, 15(5):825–838, 2005.

9 Gerhard Gentzen. Untersuchungen über das logische Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934–1935. English translation in: The Collected Papers of Gerhard Gentzen, M.E. Szabo (ed.), North-Holland 1969.

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

11 Jean-Yves Girard. Geometry of interaction 2: Deadlock-free algorithms. In *International Conference on Computer Logic*, pages 76–93, 1988.

12 Jean-Yves Girard. Proof-nets: the parallel syntax for proof-theory. *Logic and Algebra*, pages 97–124, 1996.

13 Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001. doi:10.1017/S096012950100336X.

14 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

15 Willem Heijltjes. Proof nets for additive linear logic with units. In *IEEE 26th Annual Symposium on Logic in Computer Science (LICS)*, pages 207–216, 2011.

16 Willem Heijltjes, Dominic Hughes, and Lutz Straßburger. Intuitionistic proofs without syntax. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019.

17 Willem B. Heijltjes, Dominic J. D. Hughes, and Lutz Straßburger. Normalization without syntax. Technical Report HAL-03654060, Inria, 2022. URL: https://hal.inria.fr/hal-03654060.

18 Dominic Hughes. Proofs without syntax. *Annals of Mathematics*, 164(3):1065–1076, 2006.

19 Dominic Hughes and Willem Heijltjes. Conflict nets: efficient locally canonical mall proof nets. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

20 Dominic Hughes and Rob van Glabbeek. Proof nets for unit-free multiplicative-additive linear logic. *Transactions on Computational Logic*, 6(4):784–842, 2005.

21 Dominic J. D. Hughes. First-order proofs without syntax, 2019. arXiv preprint 1906.11236. arXiv:1906.11236.

22 Dominic J. D. Hughes, Lutz Straßburger, and Jui-Hsuan Wu. Combinatorial proofs and decomposition theorems for first-order logic. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470579.

23 Dominic J.D. Hughes. Simple free star-autonomous categories and full coherence. *Journal of Pure and Applied Algebra*, 216(11):2386–2410, 2012.

24 R.J.M. Hughes. Super-combinators: a new implementation method for applicative languages. In *ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

25 J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, 2000.

**26** Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 190–203, 1985.

**27** Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 95–108, 1990.

**28** Stéphane Lengrand. *Normalisation and equivalence in proof theory and type theory.* PhD thesis, University of St. Andrews, 2006.

**29** Simon L. Peyton-Jones. *The implementation of functional programming languages.* Prentice Hall, 1987.

**30** Andrea Aler Tubella and Lutz Straßburger. Introduction to deep inference. Lecture notes for ESSLLI'19, 2019. URL: `https://hal.inria.fr/hal-02390267`.

# Decision Problems for Linear Logic with Least and Greatest Fixed Points

**Anupam Das** ✉
University of Birmingham, UK

**Abhishek De** ✉
IRIF, CNRS, Université Paris Cité & INRIA, France

**Alexis Saurin** ✉
IRIF, CNRS, Université Paris Cité & INRIA, France

**Abstract**

Linear logic is an important logic for modelling resources and decomposing computational interpretations of proofs. Decision problems for fragments of linear logic exhibiting "infinitary" behaviour (such as exponentials) are notoriously complicated. In this work, we address the decision problems for variations of linear logic with fixed points ($\mu$MALL), in particular, recent systems based on "circular" and "non-wellfounded" reasoning. In this paper, we show that $\mu$MALL is undecidable.

More explicitly, we show that the general non-wellfounded system is $\Pi^0_1$-hard via a reduction to the non-halting of Minsky machines, and thus is strictly stronger than its circular counterpart (which is in $\Sigma^0_1$). Moreover, we show that the restriction of these systems to theorems with only the least fixed points is already $\Sigma^0_1$-complete via a reduction to the reachability problem of alternating vector addition systems with states. This implies that both the circular system and the finitary system (with explicit (co)induction) are $\Sigma^0_1$-complete.

**2012 ACM Subject Classification** Theory of computation → Linear logic; Theory of computation → Proof theory; Theory of computation → Complexity theory and logic

**Keywords and phrases** Linear logic, fixed points, decidability, vector addition systems

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.20

**Related Version** *Full Version*: `https://hal.archives-ouvertes.fr/hal-03655651`

**Funding** *Anupam Das*: This author is supported by a UKRI Future Leaders Fellowship, *Structure vs. Invariants in Proofs*, project reference MR/S035540/1.
*Abhishek De*: This author has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 754362.
*Alexis Saurin*: This author has been partially supported by ANR project *RECIPROG*, project reference ANR-21-CE48-019-01.

**Acknowledgements** We would like to thank anonymous reviewers for their valuable comments that enhanced the clarity and presentation of this paper. We also thank Sylvain Schmitz for his insights on alternating vector addition systems.

## 1 Introduction

Fixed point theory occurs in just about every field of computer science, including program analysis [30], game theory [10, 46], automata theory [33, 47], and programming language theory [53]. In the setting of fixed point logics, the (multi)modal $\mu$-calculus (the extension of basic modal logic K with least and greatest fixed point operators) is probably the most well-studied. The most important result in this direction is the obtention of completeness of Hilbert-style axiomatisations for the logic [33, 59, 58]. Another relevant case study is that of

© Anupam Das, Abhishek De, and Alexis Saurin;
licensed under Creative Commons License CC-BY 4.0

Kleene Algebra (and extensions) where basic algebraic structures are extended by a "Kleene star" modelling iteration. Such theories have similarly received axiomatisations that have been proved complete (over relational and language models) [32, 35, 9, 22].

These formalisations employ inference rules that express an *explicit* (co)induction scheme *i.e.* the induction invariant must be provided explicitly. However, more recently both these settings have received proof-theoretic developments allowing for an implicit treatment of (co)induction, by way of "non-wellfounded" and "circular" reasoning [48, 1, 16]. Such systems admit greater proof-theoretic expressivity while, at the same time, reinforcing connections between these logics and automata theory. Evidence of their utility has been duly provided in recent works that recover the aforementioned completeness results using entirely proof-theoretic (as opposed to automata theoretic) methods, in particular [1], building on [48], in the case of the $\mu$-calculus, and [15], building on [16], in the case of Kleene algebra.

In a parallel direction, the extension of fragments of *linear logic* by fixed points has become increasingly developed in the last 15 years. Baelde and Miller [5, 2] developed a finitary deductive system for first-order linear logic with least and greatest fixed points. Santocanale was the first to propose a circular system in this area, in particular for an extension of "lattice logic" by fixed points in [51], and later with Fortier proved a form of cut-elimination [25]. Baelde, Doumane and Saurin in [4] extended both the system and the cut-elimination result to the full propositional fragment of Baelde and Miller's logic, now yielding three systems: $\mu\mathsf{MALL}^{\mathsf{ind}}$(based on explicit (co)induction), $\mu\mathsf{MALL}^{\infty}$ (based on non-wellfounded reasoning) and $\mu\mathsf{MALL}^{\omega}$ (based on circular reasoning).

In terms of expressivity, $\mu\mathsf{MALL}$ can be seen as an amalgamation of the properties of $\mu$-calculus and Kleene Algebras. Like Kleene Algebras, $\mu\mathsf{MALL}$ is also "resource-conscious" (indeed, Kleene Algebra and extensions are just fragments of a non-commutative $\mu\mathsf{MALL}$); and like the $\mu$-calculus, $\mu\mathsf{MALL}$ also allows for unrestricted interleaving of fixed points.

In this work, we study systems for $\mu\mathsf{MALL}$ in terms of proof-theoretic strength, in particular asking whether a system is *conservative* over another. A pertinent observation at this juncture is that the aforementioned techniques for the $\mu$-calculus and Kleene algebra for comparing such systems seem to break down in the more general setting of substructural logics. Indeed, in this work, we shall show that they do *not* hold *per se*, by addressing the complexity of deciding theorems of $\mu\mathsf{MALL}^{\infty}$ and $\mu\mathsf{MALL}^{\omega}$. In particular, we show that provability in $\mu\mathsf{MALL}^{\infty}$ is $\Pi^0_1$-hard, *i.e.* at least co-recursively enumerable. Our proof method is based on an encoding of Minsky machines that is inspired by a previous work of Kuznetsov [37]. Since $\mu\mathsf{MALL}^{\omega}$ is a calculus of finite (recursively checkable) proofs, and so is in $\Sigma^0_1$, this in particular implies that it proves strictly fewer theorems than $\mu\mathsf{MALL}^{\infty}$.

Our second main result is that $\mu\mathsf{MALL}^{*}$, the fragment of $\mu\mathsf{MALL}$ restricted to only least fixed points (on which all three aforementioned systems coincide), is $\Sigma^0_1$-complete and consequently, undecidable. We use Lincoln's idea of encoding alternating vector addition systems which he originally employed to prove the undecidability of full linear logic [41]. However, in the absence of exponentials, we have had to reinvent the encoding.

The resulting relationships between the systems we consider in this work are summarised in Figure 1.

**Organization of the paper.** This paper is organised as follows. In Section 2 we motivate our work from the point of view of "regularisation", i.e. the transformation of a non-wellfounded proof into a circular one. We describe the syntax and relevant properties of $\mu\mathsf{MALL}$ in Section 3 and show regularisation in the additive fragment. In Section 4.1, we prove our first main result that $\mu\mathsf{MALL}^{\infty}$ is $\Pi^0_1$-hard (and thus, in general, non-regularisable). In Section 5,

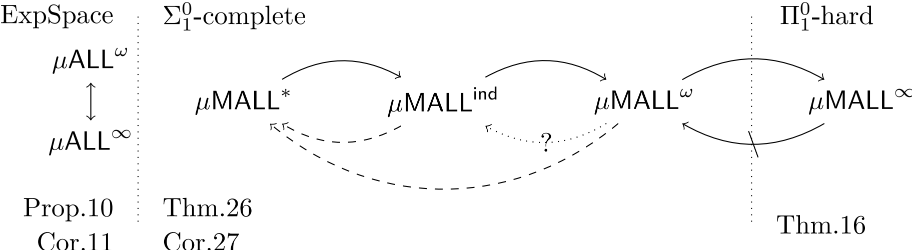



**Figure 1** Relationships between systems in this work. Solid arrows $\to$ denote inclusion, dashed arrows denote conservative extensions, negated arrows $\not\to$ denote non-inclusion.

we give our second main result, that $\mu$MALL$^*$ is $\Sigma^0_1$-complete. Finally, in Section 6, we conclude and discuss some directions of future work. Additional material, discussions and proof details can be found in an extended version of this paper [14].

## 2 Motivation: regularisation techniques are not logic-independent

Non-wellfounded systems for logics such as the $\mu$-calculus [1, 17, 21, 48], and in our case $\mu$MALL [4], handle least ("$\mu$") and greatest ("$\nu$") fixed points by identical rules:

$$\textbf{Fixed point rules:} \qquad \frac{\Gamma, \phi(\mu X.\phi(X))}{\Gamma, \mu X.\phi(X)}\,\mu \qquad\qquad \frac{\Gamma, \phi(\nu X.\phi(X))}{\Gamma, \nu X.\phi(X)}\,\nu \tag{1}$$

Here $\Gamma$ (a "sequent") is a list, set or multiset of formulas and the comma is to be read as a form of disjunction, all depending on the logic at hand. To distinguish the two fixed points, non-wellfounded proofs impose a certain global correctness condition; informally speaking, each infinite branch must have a "critical" $\nu$-formula that is unfolded infinitely often (a formal definition is given in the next section). This corresponds to a sort of "infinite descent" argument that mimics inductive reasoning on the fixed point. At least one motivation for our work is to understand when, in general, we can transform a non-wellfounded proof tree into one that is *regular*, i.e. one that has finitely many distinct subtrees, and so may be written as a finite directed (cyclic) graph.

**Regularising $\mu$-calculus is easy.** In the case of the modal $\mu$-calculus, a simple proof system is readily obtained by extending (multi)modal logic K (cf., e.g., [6]) by the rules in Equation (1). The induced (cut-free) calculus enjoys a certain generalisation of the subformula property (the "Fischer-Ladner closure") meaning that only finitely many distinct sequents may occur in a proof. As a result, once a particular sequent to be proved is fixed, the aforementioned global correctness criterion becomes an $\omega$-regular property on infinite branches. This allows us to reduce *regular* completeness of the system to *non-wellfounded* completeness of the system, thanks to Rabin's basis theorem [50]. This idea is implicit in Niwinski and Walukiewicz's seminal work [48].

This reduction is, a priori, non-constructive: it asserts the existence of a regular proof but does not tell us how to construct one from a given non-wellfounded one. However it is possible to define a constructive such procedure that "cuts" branches of an infinite proof tree to transform it into a regular one, using automata-theoretic techniques.

**Structural rules:** $\dfrac{}{\phi, \phi^\perp}\ \mathsf{id}$ $\quad \dfrac{\Gamma_1, \phi \quad \Gamma_2, \phi^\perp}{\Gamma_1, \Gamma_2}\ \mathsf{cut}$ $\quad \dfrac{\Gamma, \phi_2, \phi_1, \Delta}{\Gamma, \phi_1, \phi_2, \Delta}\ \mathsf{ex}$

**Logical rules:** $\dfrac{\Gamma, \phi_1, \phi_2}{\Gamma, \phi_1 ⅋ \phi_2}\ ⅋$ $\quad \dfrac{\Gamma_1, \phi_1 \quad \Gamma_2, \phi_2}{\Gamma_1, \Gamma_2, \phi_1 \otimes \phi_2}\ \otimes$ $\quad \dfrac{\Gamma, \phi_i}{\Gamma, \phi_1 \oplus \phi_2}\ \oplus_i$ $\quad \dfrac{\Gamma, \phi_1 \quad \Gamma, \phi_2}{\Gamma, \phi \,\&\, \phi_2}\ \&$

**Logical rules (units):** $\dfrac{}{\mathbf{1}}\ \mathbf{1}$ $\quad \dfrac{\Gamma}{\Gamma, \perp}\ \perp$ $\quad \dfrac{}{\Gamma, \top}\ \top$ $\quad$ No rule for $\mathbf{0}$

**Figure 2** Inference rules for MALL, where $i \in \{1, 2\}$.

**Regularisation not possible (in general) in predicate settings.** The situation is considerably different in predicate logics with (co)induction or fixed points, e.g. [7, 8, 54]. There neither is cut eliminable in general, nor are proofs regularisable in general, due to infinitely many choices available when instantiating existentials by terms. Indeed both of these observations have recently been demostrated formally for cyclic systems corresponding to fragments of Peano Arithmetic [13].

**The trouble with structural rules.** Returning to the propositional setting, at first glance the regularisation argument for the $\mu$-calculus is rather general, relying only on the finitude of sequents occurring in a proof to obtain $\omega$-regularity of the global correctness criterion. However, such finitude of sequents is a rather peculiar property in structural proof theory at large. For the $\mu$-calculus this is a consequence of the underlying classical framework: the admissibility of contraction (duplicating formulas) and weakening (deleting formulas) allows us to limit the number of formula repetitions in a sequent.

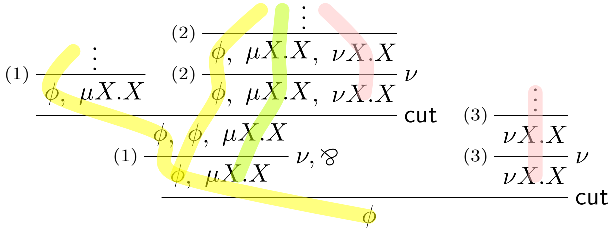
*Substructural logics* are logics lacking at least one of the usual structural rules. Decidability of substructural logics is often very difficult [34, 57, 38]. In *linear logic* [28], one of the most well-studied substructural logics, sequents are effectively multisets and the use of contraction and weakening is carefully controlled. Conjunction and disjunction each have two versions in linear logic: *multiplicative* and *additive*.

| | conjunction | disjunction | true | false |
|---|---|---|---|---|
| multiplicative | $\otimes$ | $⅋$ | $\mathbf{1}$ | $\perp$ |
| additive | $\&$ | $\oplus$ | $\top$ | $\mathbf{0}$ |

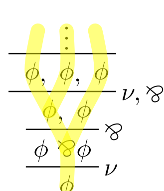
The logical system thus obtained is called multiplicative-additive linear logic (MALL) and its inference rules are depicted in Figure 2 (sequents being construed as finite lists). Note that, despite the absence of structural rules weakening and contraction, *cut-admissibility* implies that all sequents in a proof have size bounded by that of the conclusion and that the proof search space has only polynomial depth; thus provability is in PSpace (in fact, MALL is PSpace-complete [43]).

Full linear logic extends MALL by incorporating certain "exponential" modalities, written $?\phi$ and, dually, $!\phi$. Structural rules are recovered in the case of $?\phi$, and the resulting logic is undecidable [43]. This is because allowing structural rules only on certain formulas can lead to sequents of unbounded size during proof search. Notably, decidability of multiplicative exponential linear logic (MELL) is still an open question[1] [42, 55].

[1] In [43], non-commutative MELL *i.e.* the fragment without the exchange rule has been proved to be undecidable.



**(a)** A wellfounded proof.



**(b)** An unsound pre-proof.

**(c)** A circular proof.

**(d)** A non-wellfounded proof.

**Figure 3** Various shapes of proof trees for $\mu$MALL. Here $\phi = \nu X.X \parr X$. Rules marked $(i)$, for $i \in \{1, 2, 3\}$, are roots of identical subtrees.

Adding the fixed point rules from Equation (1) to MALL leads to a similar issue, and there is no general way to arrive at such a bound on the set of sequents during proof search. This not only makes decidability of provability non-trivial, but also regularisation of non-wellfounded proofs.

## 3 Preliminaries

### 3.1 $\mu$MALL: multiplicative additive linear logic with fixed points

In this subsection we recall the system $\mu\mathsf{MALL}^\infty$ introduced in [4].

▶ **Definition 1.** *Fix a countable set of propositional constants $\mathcal{A} = \{A, B, \dots\}$ and variables $\mathcal{V} = \{X, Y, \dots\}$ such that $\mathcal{A} \cap \mathcal{V} = \emptyset$.* $\mu$MALL **pre-formulas** *are given by the grammar:*

$$\phi, \psi ::= \mathbf{0} \mid \top \mid \bot \mid \mathbf{1} \mid A \mid A^\perp \mid X \mid \phi \parr \psi \mid \phi \otimes \psi \mid \phi \oplus \psi \mid \phi \,\&\, \psi \mid \mu X.\phi \mid \nu X.\phi$$

*where $A \in \mathcal{A}$, $X \in \mathcal{V}$, and $\mu, \nu$ bind the variable $X$ in $\phi$. Free and bound variables, and capture-avoiding substitution are defined as usual. The subformula ordering is denoted $\leq$. When a pre-formula is closed (i.e. no free variables), we simply call it a* **formula***.*

Negation, $(\bullet)^\perp$, defined as a meta-operation on pre-formulas, will be used only on formulas. As it is not part of the syntax, we do not need any positivity condition on the fixed-point expressions. As expected, least and greatest fixed-points are the dual of each other.

▶ **Definition 2.** **Negation** *of a pre-formula is defined inductively as follows.*

$$(\mathbf{0})^\perp = \top; \quad (\top)^\perp = 0; \quad (\bot)^\perp = \mathbf{1}; \quad (\mathbf{1})^\perp = \bot; \quad (A)^\perp = A^\perp; \quad (A^\perp)^\perp = A;$$
$$(X)^\perp = X; \quad (\phi \parr \psi)^\perp = \phi^\perp \otimes \psi^\perp; \quad (\phi \otimes \psi)^\perp = \phi^\perp \parr \psi^\perp; \quad (\phi \oplus \psi)^\perp = \phi^\perp \,\&\, \psi^\perp;$$
$$(\phi \,\&\, \psi)^\perp = \phi^\perp \oplus \psi^\perp; \qquad (\mu X.\phi)^\perp = \nu X.\phi^\perp; \qquad (\nu X.\phi)^\perp = \mu X.\phi^\perp.$$

The system is classical, hence, it is enough to consider a one-sided proof system. However, as discussed in Section 2 it is imperative to allow multiple copies of the same formula in a sequent. A one-sided $\mu$MALL sequent is thus a finite list of formulas.

▶ **Definition 3.** *A* **pre-proof** *of* $\mu$MALL$^\infty$ *is a possibly infinite tree generated from the inference rules of* MALL *(see Figure 2) and the fixed point rules from Equation* (1)*.*

Let us recall some standard terminology relating to inference rules [11]. The sequent(s) in a rule displayed above the line are **premisse(s)** and the unique sequent below the line is the **conclusion**. In a logical or fixed point rule, the **principal formula** is the distinguished formula occurrence in its conclusion in Equation (1) or Figure 2. **Auxiliary formulas** are the formula occurrences distinguished in the premisse(s). Other formula occurrences in logical or fixed point rules are **side formulas**.

▶ **Definition 4.** *Given a pre-proof* $\pi$*, for all rules* $r$ *occurring in* $\pi$*, we define the* **immediate ancestor** *relation* $\mathsf{IA}(r)$ *on formula occurrences of* $r$ *by:* $(\phi, \psi) \in \mathsf{IA}(r)$ *if* $\phi$ *is principal and* $\psi$ *is auxiliary; or* $\phi$ *is a side formula occurrence in the conclusion and* $\psi$ *is the corresponding side formula occurrence in a premisse; or* $r$ *is structural and* $\phi$ *is a formula occurrence in the conclusion and* $\psi$ *is the corresponding formula occurrence in a premisse.*

Several examples of pre-proofs can be found in Figure 3. Immediate ancestors are indicated by the same colour (note that immediate ancestors always "go upwards").

One of the key caveats of non-wellfounded pre-proofs is that, unconstrained, they admit inconsistencies: it is possible to derive any sequent, as shown in Figure 3b. For this reason we impose a global criterion on pre-proofs.

▶ **Definition 5** ([4])**.** *Let* $\beta = (\Gamma_i)_{i<\omega}$ *be an infinite branch of a* $\mu$MALL$^\infty$ *pre-proof* $\pi$ *and let* $r_i$ *be the rule with conclusion* $\Gamma_i$*. A* **thread** *of* $\beta$ *is given by* $k \in \mathbb{N}$ *and a sequence of formula occurrences* $\{\phi_i\}_{k<i<\omega}$ *such that, for* $k < i < \omega$*, we have* $(\phi_i, \phi_{i+1}) \in \mathsf{IA}(r_i)$*.*

*A thread* $\tau$ *is* **progressing** *if: it is infinitely often principal; and, the smallest formula occurring infinitely often in* $\tau$ *is a* $\nu$*-formula.*[2]

$\pi$ *is called a* **proof** *if every infinite branch has a progressing thread.*

For example, in Figure 3b, while the right infinite branch has a progressing thread along $\nu X.X$ (indicated red), the left branch has no progressing thread, so the pre-proof is not a proof. Figure 3d is indeed a proof, assuming, say, each $\nu$ step has the left-most $\phi$ occurrence principal. In this work we shall crucially make use of the *admissibility* of cut in $\mu$MALL$^\infty$:

▶ **Theorem 6** ([4, 3, 19])**.** *Every provable* $\mu$MALL$^\infty$ *sequent has a cut-free proof.*

Finally, we consider a fragment of pre-proofs that have a finite presentation.

▶ **Definition 7.** *A* $\mu$MALL$^\infty$ *pre-proof is said to be* **circular** *(aka* **regular***) if it has finitely many distinct sub-trees.The class of circular proofs is denoted by* $\mu$MALL$^\omega$*.*

Figure 3c is a regular pre-proof. In fact, it is a proof; any infinite branch must either loop on one of (1), (2) or (3), whence there is an infinite progressing thread on $\phi$ (indicated yellow), $\nu X.X$ (indicated red) or $\nu X.X$ (indicated red) respectively, or it alternates between (1) and (2) infinitely often, whence there is an infinite progressing thread (indicated yellow) on $\phi$.

Importantly, given a regular pre-proof $\pi$, we can decide whether it is a proof by reduction to the universality of non-deterministic parity $\omega$-word automata, cf. [48, 18, 21]. Observe that $\nu$-unfoldings are the source of infiniteness in proofs: with only $\mu$-unfoldings, no infinite branch may have a progressing thread. So $\nu$-free proofs, *i.e.* proofs without any $\nu$s, are necessarily finite. Let us call this class of proofs $\mu$MALL$^*$; clearly, $\mu$MALL$^* \subseteq \mu$MALL$^\omega \subseteq \mu$MALL$^\infty$.

[2] A "smallest" formula must exist along a thread, since immediate ancestry is compatible with the Fischer-Ladner pre-order, cf. [24] (see also [56, 21]). By construction this formula is unique and, furthermore, is a subformula of all other infinitely occurring formulas in $\tau$.

$$\dfrac{\Gamma,\psi \qquad \bullet\,\dfrac{\psi^{\perp},\phi(\psi) \qquad \dfrac{\dfrac{\bullet\,\dfrac{\vdots}{\psi^{\perp},\nu X.\phi(X)}\,\mathsf{cut}}{\phi^{\perp}(\psi^{\perp}),\phi(\nu X.\phi(X))}\,\phi}{\phi^{\perp}(\psi^{\perp}),\nu X.\phi(X)}\,\nu}{\psi^{\perp},\nu X.\phi(X)}\,\mathsf{cut}}{\nu X.\phi(X),\Gamma}\,\mathsf{cut}$$

**Figure 4** A (logic-independent) simulation of the Park's rule in circular proofs. The steps marked $\phi$ are given by "functoriality" or "deep inference" with respect to the positive formula $\phi(X)$.

Although the focus of the paper are these systems, we briefly discuss $\mu\mathsf{MALL}^{\mathsf{ind}}$, the wellfounded system with explicit coinduction. The system has the same set of inference rules as Definition 3 except the rule for the greatest fixed-point which is replaced by the so-called *Park's rule*, implementing a form of (co)induction:

$$\dfrac{\Gamma,\psi \quad \psi^{\perp},\phi(\psi)}{\Gamma,\nu X.\phi(X)}\,\nu_{\mathsf{prks}}$$

We exhibit a $\mu\mathsf{MALL}^{\mathsf{ind}}$ proof in Figure 3a. We still have that $\mu\mathsf{MALL}^{*} \subseteq \mu\mathsf{MALL}^{\mathsf{ind}}$, and it is not hard to see that $\mu\mathsf{MALL}^{\mathsf{ind}} \subseteq \mu\mathsf{MALL}^{\omega}$ as shown in Figure 4.

The opposite direction *i.e.* the question of equiprovability of $\mu\mathsf{MALL}^{\mathsf{ind}}$ and $\mu\mathsf{MALL}^{\omega}$ is a manifestation of the so-called *Brotherston-Simpson conjecture* in the setting of $\mu\mathsf{MALL}$ [8] and is a difficult open question.

## 3.2 Focusing

In structural proof theory, focused proofs are a family of proofs that have more structure than usual sequent calculus proofs. The additional structure brought by focusing will be crucial in the next sections in order to extract traces of execution in the computational models that we consider. We describe focused proofs as a complete, proper class of $\mu\mathsf{MALL}^{\infty}$ proofs. The starting point of focusing is the classification of the inference rules (*resp.* connectives) of linear logic into two categories: **positive** and **negative**.

The **negative** connectives have ***invertible*** inferences: if the conclusion of the inference is provable, so are its premisses. For example, if a sequent $\Gamma, \phi \parr \psi$ is provable, so is $\Gamma, \phi, \psi$. The negative (resp. positive) connectives of $\mu\mathsf{MALL}^{\infty}$ are $\&, \parr, \bot, \top, \nu$ (resp. $\otimes, \oplus, 1, 0, \mu$).[3]

By assigning arbitrary polarities to atomic variables one can extend the notion to formulas in such a way that each formula is either positive or negative. A sequent is ***positive*** if it contains only positive or atomic formulas, it is ***negative*** otherwise.

▶ **Definition 8.** *A $\mu\mathsf{MALL}^{\infty}$ proof is said to be in* **negative normal form** *if every negative sequent occurring in it is the conclusion of a negative inference. A $\mu\mathsf{MALL}^{\infty}$ proof $\pi$ is said to be* **focused** *if it is in negative normal form and if, for every rule r with a positive sequent s as conclusion, the auxiliary formulas of r are principal in its premisses (the "focus"), unless they are negative atomic formulas.*[4]

[3] Observe that both the $\mu$ and $\nu$ rules are invertible. See [4, 21] for an explanation of the choice.
[4] As is usual, we neglect the structural rule of exchange in this definition, by working with the exchange built in the other rules.

Note that the focusing constraint enforces that when a positive formula is principal (the "focus"), so are its auxiliary formulas and so on until a negative formula is reached.

▶ **Theorem 9** ([4, 21]). *If a sequent is provable in $\mu\mathsf{MALL}^{\infty}$, it has a focused cut-free proof.*[5]

## 3.3 Regularising fragments of $\mu$MALL$^{\infty}$: the complexity of $\mu$ALL

While cuts are admissible in $\mu\mathsf{MALL}^{\infty}$, cf. Theorem 6, regularity of proofs is not, in general, preserved by cut-elimination. In other words, the process of cut-elimination on a circular proof produces a non-wellfounded proof which, in general, may not have finitely many distinct sub-proofs. In fact, we can show that the cut-free $\mu\mathsf{MALL}^{\infty}$ and the cut-free $\mu\mathsf{MALL}^{\omega}$ are not equiprovable since $\nu X.X \parr X$ has a unique cut-free $\mu\mathsf{MALL}^{\infty}$ proof (up to choices of principal formulas), given in Figure 3d, that is non-regular. However, there is indeed a circular proof with cuts (see Figure 3c) of the aforementioned theorem. It is natural to ask: *Is every theorem of $\mu\mathsf{MALL}^{\infty}$ also provable in $\mu\mathsf{MALL}^{\omega}$ (possibly with cuts)?* In this paper we formally show that such a regularisation result does not hold.

It is worth pointing out that the argument we mentioned for regularisation in the $\mu$-calculus in Section 2 can in fact be adapted to certain fragments of $\mu$MALL, in particular the additive fragment. Writing $\mu\mathsf{ALL}^{\infty}$ and $\mu\mathsf{ALL}^{\omega}$ for the restriction of $\mu\mathsf{MALL}^{\infty}$ and $\mu\mathsf{MALL}^{\omega}$, respectively, to only additive connectives, we have:

▶ **Proposition 10.** *If $\Gamma$ is provable in $\mu\mathsf{ALL}^{\infty}$, then it is also (cut-free) provable in $\mu\mathsf{ALL}^{\omega}$.*

**Proof.** By the cut-elimination theorem of [25], we may assume that $\Gamma$ has a cut-free $\mu\mathsf{ALL}^{\infty}$ proof $\pi$. Note that each (non-cut) rule of $\mu$ALL preserves, bottom-up, the number of formulas in a sequent. Since there are only finitely many formulas that can occur (just those in the Fischer-Ladner closure of $\Gamma$, cf. [4]), $\pi$ may contain only finitely many distinct sequents.

As a result, the set of non-wellfounded proofs of $\Gamma$ constitutes an $\omega$-regular tree language (since the progressing thread criterion is $\omega$-regular). Since we assumed that this language was non-empty, there must be a regular such proof by Rabin's basis theorem [50]. ◀

Note that this also implies the *decidablility* of $\mu\mathsf{ALL}^{\infty}$ since, after guessing a (exponential-size) pre-proof of $\Gamma$, checking that it is a proof is decidable (in space polynomial in the size of the proof).

▶ **Corollary 11.** *$\mu\mathsf{ALL}^{\infty}$ (equivalently $\mu\mathsf{ALL}^{\omega}$) is decidable in exponential space.*

We stop short of attempting to optimise this result since, in particular, it seems sensitive to the precise presentation of $\mu$ALL. Often ($\mu$)ALL is presented with *exactly* two formulas in a sequent, e.g. [51, 25], and this invariant is maintained by the rules of ($\mu$)ALL. In such a presentation, there are only quadratically many distinct sequents in a $\mu\mathsf{ALL}^{\infty}$ proof.

However note that the calculi $\mu\mathsf{ALL}^{\omega}$ and $\mu\mathsf{ALL}^{\infty}$ make sense with an arbitrary number of formulas in a sequent, since branches need not terminate at an initial step. For instance, it is easy to see that $\mu\mathsf{ALL}^{\infty}$ proves $\Gamma, \nu X.X$, for any $\Gamma$, by simply continuously unfolding $\nu X.X$. In this more general setting the number of possible sequents becomes exponential.

[5] The focusing result in [4] is for a logic without atoms but the proof technique can be straightforwardly extended to account for atoms.

# 4 $\mu$MALL$^\infty$ is $\Pi^0_1$-hard via (non-halting of) Minsky machines

We prove that the following problem is undecidable by a reduction to the non-halting of Minsky machines.

> Given a sequent $\Gamma$ does there exist a $\mu$MALL$^\infty$ proof of $\Gamma$?

Our reduction is inspired by [37] for commutative action logic with Kleene star. At first glance, it seems straightforward to be able to embed this logic in $\mu$MALL$^\infty$ via the standard encoding of the Kleene star as $F^* = \mu X.(1 \oplus (F \otimes X))$. However, there are a couple of issues with this.

First, action logic is intuitionistic, requiring an extension of the conservativity of linear logic over intuitionistic linear logic [52] to $\mu$MALL$^\infty$. Strictly speaking, this is not possible since $\mathbf{0}$ is itself encodable as a fixed point *viz.* $\mu X.X$, and it is not obvious what language such a conservativity result might hold over.

Moreover, the inference rule for the Kleene star in [37] is omega-branching. Therefore, one would also need to establish translations from the omega-branching $\mu$MALL to $\mu$MALL$^\infty$ (and vice versa) which seem to be quite nontrivial and require yet further intermediary systems. Therefore, we provide a direct reduction.

## 4.1 The hardness result

We begin by formally defining a Minsky machine and its corresponding (non)-halting problem.

▶ **Definition 12.** *A* **Minsky machine** $\mathcal{M}$ *is a tuple* $(Q, r_1, r_2, I)$ *where* $Q$ *is a finite set of states,* $r_1, r_2$ *are two registers, and* $I$ *is a set of instructions of the form* $\mathsf{INC}(\bullet,\bullet,\bullet)$ *and* $\mathsf{JZDEC}(\bullet,\bullet,\bullet,\bullet)$ *that manipulate the current state and the contents of the registers. The operational semantics of* $\mathcal{M}$ *is given by its configuration graph, the vertices of which are configurations of form* $\langle q,a,b\rangle \in Q \times \mathbb{N} \times \mathbb{N}$ *and edges are one of the following forms:*

$$\langle p,a,b\rangle \xrightarrow{\mathsf{INC}(p,r_1,q)} \langle q,a+1,b\rangle \qquad \langle p,a,b\rangle \xrightarrow{\mathsf{INC}(p,r_2,q)} \langle q,a,b+1\rangle$$
$$\langle p,0,b\rangle \xrightarrow{\mathsf{JZDEC}(p,r_1,q_0,q_1)} \langle q_0,0,b\rangle \qquad \langle p,a,0\rangle \xrightarrow{\mathsf{JZDEC}(p,r_2,q_0,q_1)} \langle q_0,a,0\rangle$$
$$\langle p,a+1,b\rangle \xrightarrow{\mathsf{JZDEC}(p,r_1,q_0,q_1)} \langle q_1,a,b\rangle \qquad \langle p,a,b+1\rangle \xrightarrow{\mathsf{JZDEC}(p,r_2,q_0,q_1)} \langle q_1,a,b\rangle$$

*Given a state* $q_s$*, a* **run** *of* $\mathcal{M}$ *is a sequence of configurations* $\{s_i\}_{i\in\mathsf{o}}$ $(\mathsf{o} \in \omega+1)$ *such that* $s_0 = \langle q_s,0,0\rangle$ *and for all* $i \in \mathsf{o}$ *with* $i+1 \in \mathsf{o}$, $(s_i, s_{i+1})$ *is an edge in the configuration graph.*

▶ **Theorem 13** ([45]). *Given a Minsky machine* $\mathcal{M}$ *and an initial state* $q_s$*, checking that it has an infinite run from* $q_s$ *is* $\Pi^0_1$*-hard.*

Fixing a Minsky machine as in the definition above, we construe $\{a,b,z_a,z_b\} \cup Q$ as a set of propositional variables (assuming $\{a,b,z_a,z_b\} \cap Q = \emptyset$). We use $a$ and $z_a$ (*resp.* $b$ and $z_b$) to represent the contents of the register $r_1$ (*resp.* $r_2$). We encode instructions (with any extra 0-ary instruction `zero-check`) as follows:

$$[\mathsf{INC}(p,r_1,q)] \triangleq p \parr (q^\perp \otimes a^\perp)$$
$$[\mathsf{JZDEC}(p,r_1,q_0,q_1)] \triangleq (p \parr (q_0^\perp \oplus z_a^\perp)) \mathbin{\&} ((p \parr a) \parr q_1^\perp)$$
$$[\texttt{zero-check}] \triangleq (z_a \otimes z_a^\perp) \oplus (z_b \otimes z_b^\perp)$$

For any formula $F$, define $F^* = \mu X.(\mathbf{1} \oplus (F \otimes X))$ and $F^\omega = \nu X.(\bot \,\&\, (F \parr X))$. Observe that $(F^*)^\perp = (F^\perp)^\omega$. For typographic ease, we use $a^n$ to denote $\overbrace{a, \ldots, a}^{n \text{ times}}$ in a sequent.

▶ **Proposition 14.** *For any formula $F$ and any $n \in \mathbb{N}$, $F^n, (F^\perp)^*$ is provable.*

**Proof.** We proceed by induction on $n$. We call $\pi_F^n$ the proof of $F^n, (F^\perp)^*$.

**Base Case:** $n = 0$. We have

$$\dfrac{\dfrac{\dfrac{}{\mathbf{1}}\ \mathbf{1}}{\mathbf{1} \oplus (F^\perp \otimes (F^\perp)^*)}\ \oplus_1}{(F^\perp)^*}\ \mu$$

**Induction Case:** $n = m + 1$. We have

$$\dfrac{\dfrac{\dfrac{}{F, F^\perp}\ \mathsf{id} \qquad \dfrac{\mathrm{IH} = \pi_F^m}{F^m, (F^\perp)^*}}{F^{m+1}, F^\perp \otimes (F^\perp)^*}\ \otimes}{F^{m+1}, (F^\perp)^*}\ \mu, \oplus_2$$

◀

In the following, let $S$ be a finite set and $[\bullet] : S \to \mu\mathsf{MALL}^\infty$. We write $\mathsf{CH}_S$ for $\bigoplus_{s \in S} [s]^\perp$, the formula that offers a choice of picking the dual of one of the (encoding of) elements of $S$.

When $S$ is a set of instructions we rely on the above encoding, when $S$ is a set of states, we use the identity encoding benefiting from the fact that states are indeed propositional variables. The reader might be surprised by our use of the logical duality here: it is simply because we are working in the one-sided calculus. Finally, we encode the invariant to be maintained by

$$\boxed{\mathsf{Inv} \triangleq ((a^\perp)^* \otimes (b^\perp)^* \otimes \mathsf{CH}_Q) \oplus ((b^\perp)^* \otimes z_a) \oplus ((a^\perp)^* \otimes z_b).}$$

It checks one of the three following conditions: (i) the control is at a valid configuration (ii) $r_1$ is zero (iii) $r_2$ is zero. Note that $[q] = q$ where the left-hand side is the state $q$ and the right-hand side is the propositional variable $q$.

▶ **Theorem 15.** *A Minsky machine $\mathcal{M}$ has an infinite run from the state $q_s$ iff $\mathsf{CH}_I^\omega, q_s, \mathsf{Inv}$ is derivable in $\mu\mathsf{MALL}^\infty$.*

As a direct consequence of Theorem 15 and Theorem 13 we have the following:

▶ **Theorem 16.** *The set of $\mu\mathsf{MALL}^\infty$-provable sequents is $\Pi_1^0$-hard.*

The main technical ingredient of Theorem 15 is the following lemma.

▶ **Lemma 17.** *$\mathcal{M}$ performs $n$ steps starting from $\langle q_s, 0, 0\rangle$ iff $\mathsf{CH}_I^n, q_s, \mathsf{Inv}$ is derivable.*

Before showing how this lemma is proved, let us first see how it allows us to obtain our main result:

**Proof of Theorem 15.** For the only if part we assume that $\mathcal{M}$ loops. So, $\mathcal{M}$ runs for $n$ steps for all $n \in \mathbb{N}$. Therefore, by Lemma 17, we have that $\Gamma_n = \mathsf{CH}_I^n, q_s, \mathsf{Inv}$ is derivable for all $n \in \mathbb{N}$. Let us call $\pi_n$ a proof of $\Gamma_n$, for $n \in \mathbb{N}$. We have

$$\dfrac{\dfrac{\dfrac{\pi_0}{q_s, \mathsf{Inv}}}{\bot, q_s, \mathsf{Inv}}\ \bot \qquad \dfrac{\dfrac{\dfrac{\pi_1}{\mathsf{CH}_I, q_s, \mathsf{Inv}}}{\mathsf{CH}_I, \bot, q_s, \mathsf{Inv}}\ \bot \qquad \dfrac{\vdots}{\mathsf{CH}_I, \mathsf{CH}_I^\omega, q_s, \mathsf{Inv}}}{\mathsf{CH}_I \parr \mathsf{CH}_I^\omega, q_s, \mathsf{Inv}}\ \parr, \nu, \&}{\mathsf{CH}_I^\omega, q_s, \mathsf{Inv}}\ \nu, \&$$

$$
\dfrac{
\dfrac{}{p^\perp, p}\,\mathsf{id}
\qquad
\dfrac{
\dfrac{\mathrm{IH} = \pi_F^m}{q_0, \mathsf{CH}_I^\ell, b^n, \mathsf{Inv}}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{z_a, z_a^\perp}\,\mathsf{id}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{\pi_b^n}{b^n, (b^\perp)^*}
\qquad
\dfrac{}{z_a, z_a^\perp}\,\mathsf{id}
}{z_a, b^n, (b^\perp)^* \otimes z_a^\perp}\,\otimes
}{z_a, b^n, \mathsf{Inv}}\,\oplus_2
\atop \vdots
}{z_a, \mathsf{CH}_I^{\ell-1}, b^n, \mathsf{Inv}}
}{z_a, z_a \otimes z_a^\perp, \mathsf{CH}_I^{\ell-1}, b^n, \mathsf{Inv}}\,\otimes
}{z_a, \texttt{zero-check}, \mathsf{CH}_I^{\ell-1}, b^n, \mathsf{Inv}}\,\oplus_1
}{z_a, \mathsf{CH}_I^\ell, b^n, \mathsf{Inv}}\,\oplus
}{(q_0 \,\&\, z_a), \mathsf{CH}_I^\ell, b^n, \mathsf{Inv}}\,\&
}{
\dfrac{
\dfrac{p^\perp \otimes (q_0 \,\&\, z_a), \mathsf{CH}_I^\ell, p, b^n, \mathsf{Inv}}{[\mathsf{JZDEC}(p, r_1, q_0, q_1)]^\perp, \mathsf{CH}_I^\ell, p, b^n, \mathsf{Inv}}\,\oplus_1
}{\mathsf{CH}_I^{\ell+1}, p, b^n, \mathsf{Inv}}\,\oplus
}\,\otimes
$$

**Figure 5** A proof that does a zero-check.

Observe that this pre-proof is indeed a proof as the right-most non-wellfounded branch is validated by a thread on $\mathsf{CH}_I^\omega$. For the other direction assume that we have a proof $\pi$ of $\mathsf{CH}_I^\omega, q_s, \mathsf{Inv}$. Observe that for all $n \in \mathbb{N}$ we have a proof of $\mathsf{CH}_I^n, q_s, \mathsf{Inv}$:

$$
\dfrac{
\dfrac{\pi_{\mathsf{CH}_I}^n}{\mathsf{CH}_I^n, (\mathsf{CH}_I^\perp)^*}
\qquad
\dfrac{\pi}{\mathsf{CH}_I^\omega, q_s, \mathsf{Inv}}
}{\mathsf{CH}_I^n, q_s, \mathsf{Inv}}\,\mathsf{cut}
$$

By Lemma 17, $\mathcal{M}$ runs at least $n$ steps for all $n \in \mathbb{N}$. We collect all these runs and get a finitely branching infinite tree rooted at $\langle q_s, 0, 0\rangle$. König's lemma ensures that there is an infinite run of $\mathcal{M}$ from $q_s$. ◀

**Proof sketch of Lemma 17.** We will prove a stronger statement (stated this way it is easier to apply induction, however, as demonstrated above, we only need the weaker statement to prove the theorem): $\mathcal{M}$ performs $k$ steps from $\langle p, m, n\rangle$ iff $\mathsf{CH}_I^k, p, a^m, b^n, \mathsf{Inv}$ is derivable.

The **only-if** part is proved by induction on $k$. The base case ensures that the initial configuration is indeed a valid configuration. For the induction case, one examines the first step of the execution and applies the corresponding encoding of the instruction. We will exhibit the case of the decrementation of a zero-valued register.

Suppose the first instruction is $\mathsf{JZDEC}(p, r_1, q_0, q_1)$ and $m = 0$. We have the derivation shown in Figure 5 where we select the appropriate instruction by applying the corresponding $\oplus$ inference. The vertical ellipsis symbolises the repeating pattern decreasing the number of $\mathsf{CH}_I$ formulas in the sequent.

For the **if** part we first observe that the proof is necessarily finite and hence we can induct on it. The base case is vacuous. For the induction case, we will first assume (*wlog* by Theorem 6 and Theorem 9) that we have a focused and cut-free proof of $\mathsf{CH}_I^k, p, a^m, b^n, \mathsf{Inv}$. We assign atomic polarities as follows: $a, b$ and $q$ are negative for any state $q \in Q$, $z_a, z_b$ are positive. By careful case-analysis, we get that one of the $\mathsf{CH}_I$s is necessarily the focus. The instruction it chooses, we will execute that on $\mathcal{M}$. Finally, we check that `zero-check` cannot be chosen and after choosing a decrementation one cannot be led astray into the wrong state.

Basically a focused proof is forced to go exactly as exhibited in the only-if part and we will end up in a subproof of the shape $\mathsf{CH}_I^{k-1}, q, a^{m'}, b^{n'}, \mathsf{Inv}$ for some state $q$ and some natural number $m', n'$. We can then apply the induction hypothesis and get the desired result. ◀

## 4.2 Separation of regular and non-wellfounded proofs

An immediate consequence of our results is that $\mu\mathsf{MALL}^\omega$ and $\mu\mathsf{MALL}^\infty$ are distinct logics.

▶ **Theorem 18.** *There are theorems of $\mu\mathsf{MALL}^\infty$ that are not provable in $\mu\mathsf{MALL}^\omega$.*

**Proof.** Any $\mu\mathsf{MALL}^\omega$ pre-proof has only finitely many distinct sequents, and so can be checked for correctness recursively by reduction to universality of non-deterministic parity automata over infinite words. Thus $\mu\mathsf{MALL}^\omega \in \Sigma_1^0$. On the other hand, we showed in Theorem 16 that $\mu\mathsf{MALL}^\infty$ is $\Pi_1^0$-hard, and we conclude since $\Pi_1^0 \setminus \Sigma_1^0 \neq \emptyset$. ◀

Observe that this proof is apparently non-constructive in the sense that we do not explicitly exhibit a sequent in $\mu\mathsf{MALL}^\infty \setminus \mu\mathsf{MALL}^\omega$. While it is clear that not all sequents of the form $\mathsf{CH}_I^\omega, q_s, \mathsf{Inv}$ from Section 4 can be derivable in $\mu\mathsf{MALL}^\omega$, it is not clear which particular Minsky machine $\mathcal{M}$ to choose to witness this underivability. In fact the argument can indeed be constructivised using established recursion-theoretic techniques, namely the notion of *productive function*. The application of such techniques to the present situation is explained elegantly by Kuznetsov in [[36], pp. 497], so we shall not recast it here.

# 5 $\mu\mathsf{MALL}^*$ is $\Sigma_1^0$-complete via (reachability in) AVASS

## 5.1 Towards an upper bound

The works of Palka [49] and Kuznetsov [37] proceed by showing $\Pi_1^0$ membership of their various logics (say $L$) in two stages:

1. The cut-free fragment $L^\mu$ with only least fixed points (*i.e.* the Kleene star is only on the right side of the sequent) is decidable.
2. The provability problem for any sequent is in $\Pi_1^0(L^\mu)$, whence it is in $\Pi_1^0$ by (1) above.

Usually, the difficult part is (2), requiring some combination of proof-theoretic and logical techniques, typically requiring infinitary wellfounded proof search to obtain the $\Pi_1^0$ bound. However, in our case, we are already stuck at (1): $\mu$-only cut-free $\mu\mathsf{MALL}$, i.e. $\mu\mathsf{MALL}^*$, is undecidable. In the absence of greatest fixed-points, all systems (non-wellfounded, circular, inductive) coincide; so the logic $\mu\mathsf{MALL}^*$ is indeed an interesting and robustly defined core of the theory of linear logic with fixed points.

Propositional linear logic was shown to be undecidable [41, 43] by a reduction from the reachability problem in an *and-branching* two counter machine without zero-test. Such machines are essentially equivalent to a particular extension of vector addition systems, called *alternating vector addition systems with states* (or AVASS) [39, 31] (in particular, the fork rule is exactly the same). More recently, other substructural logics have been related with extensions of vector addition systems [20, 39, 40]. Our work is in the spirit of this line of research and we show the undecidability of $\mu\mathsf{MALL}^*$ by a reduction from the reachability problem of AVASS.

One could try using the undecidability of propositional linear logic to prove the undecidability of $\mu\mathsf{MALL}^*$ using a standard encoding of the exponential modalities by fixed point formulas of the following form:

$$[?F] = \mu X.\bot \oplus [F] \oplus (X \parr X) \quad ; \quad [!F] = \nu X.\mathbf{1} \,\&\, [F] \,\&\, (X \otimes X)$$

However, this encoding is not known to be faithful. Note that the reduction in [41, 43] uses only ? so the encoding is indeed in $\mu\mathsf{MALL}^*$. We provide a direct proof via the reachability problem of AVASS.

Before defining the reduction in the next subsection (Section 5.2), we conclude this subsection by formally introducing AVASS and the corresponding reachability problem.

▶ **Definition 19.** *An* **AVASS** *is a tuple $\mathcal{B} = (Q, Q_\ell, k, \mathsf{A}, T_u, T_f)$ such that:*

- *$Q$ is a finite set of states with $Q_\ell \subseteq Q$;*
- *$k \in \mathbb{N}$ is called the* **dimension***;*
- *$\mathsf{A}$ is a finite subset of $\mathbb{N}^k$ called the set of* **axioms***;*
- *$T_u \subseteq Q \times \mathbb{Z}^k \times Q$ and $T_f \subseteq Q^3$ are finite and called the* **unary** *and* **fork** *rules respectively.*

*A configuration of an AVASS $\mathcal{B}$ is a pair $(q, \vec{v}) \in Q \times \mathbb{N}^k$ where $Q$ is the set of states of $\mathcal{B}$ and $k$ is its dimension.*

▶ **Definition 20.** *Given an AVASS $\mathcal{B} = (Q, Q_\ell, k, \mathsf{A}, T_u, T_f)$, a configuration $(q, \vec{v}) \in Q \times \mathbb{N}^k$ is said to be* **reachable** *if there is a binary tree labelled by configurations such that:*

- *The root node is labelled by $(q, \vec{v})$.*
- *If a node $(q, \vec{v})$ has a unique child $(q', \vec{v'})$ then $(q, \vec{v} - \vec{v'}, q') \in T_u$.*
- *If a node $(q, \vec{v})$ has children $(q', \vec{v'})$ and $(q'', \vec{v''})$ then $\vec{v} = \vec{v'} = \vec{v''}$ and $(q, q', q'') \in T_f$.*
- *The leaves are labelled by elements of $Q_\ell \times \mathsf{A}$.*

*Such that a binary tree is called the* **run tree** *of the configuration.*

▶ **Theorem 21** ([43]). *The AVASS reachability problem is $\Sigma^0_1$-complete.*

## 5.2 Encoding an AVASS in $\mu\mathsf{MALL}^*$

We fix $k+1$ propositional variables, $a_1, \ldots, a_k, z$, and define below an encoding of integer vectors of dimension up to $k$ (the unique vector of dimension 0 is written $\epsilon$). For the purpose of the encoding vectors will be read from left to right *i.e.* a vector $\vec{v}$ of dimension $l+1$ will be of the form $(n, \vec{u})$ for an integer $n$ and a vector $\vec{u}$ of dimension $l$.

▶ **Definition 22.** *The* **encoding of an integer vector** *$\vec{v}$ of dimension $d$, relative to propositional variables $b_i, \ldots, b_{d+i-1}, z$, written $[\vec{v}]_{b_i,\ldots,b_{d+i-1},z}$, is defined inductively as follows:*

$$[\vec{v}]_{b_i,\ldots,b_{d+i-1}} \triangleq \begin{cases} z & \text{if } \vec{v} = \epsilon; \\ b_i \parr [\vec{v'}]_{b_i,\ldots,b_{d+i-1},z} & \text{if } \vec{v} = (n, \vec{u}), n \geq 1, \text{ and } \vec{v'} = (n-1, \vec{u}); \\ b_i^\perp \otimes [\vec{v'}]_{b_i,\ldots,b_{d+i-1},z} & \text{if } \vec{v} = (n, \vec{u}), n \leq -1, \text{ and } \vec{v'} = (n+1, \vec{u}); \\ [\vec{u}]_{b_{i+1},\ldots,b_{d+i-1},z} & \text{if } \vec{v} = (0, \vec{u}). \end{cases}$$

*We will simply write $[\vec{v}]$ for the encoding of a vector of dimension $k$ relative to $a_1, \ldots, a_k, z$. (We also use this lighter notation for vectors of lower dimension when the dimension and the $\{a_i, \ldots, a_k, z\}$ to be used are clear from the context.)*

The encoding is slightly involved, so let us first consider an example:

▶ **Example 23.** Consider the encoding of $(-1, 0, 1)$ relative to $b_1, b_2, b_3, z$.

$$\begin{aligned}[(-1,0,1)]_{b_1,b_2,b_3} &= {b_1}^\perp \otimes [(0,0,1)]_{b_1,b_2,b_3} \\ &= {b_1}^\perp \otimes [(1)]_{b_3} = {b_1}^\perp \otimes (b_3 \parr [(0)]_{b_3}) \\ &= {b_1}^\perp \otimes (b_3 \parr [\epsilon]_{b_3}) = {b_1}^\perp \otimes (b_3 \parr z)\end{aligned}$$

Observe that the $i^{\text{th}}$ coordinate is represented by the propositional variables $b_i$. The following lemma shows that the encoding is meaningful with respect to vector equality.

▶ **Lemma 24.** *Let $\vec{u}$ and $\vec{v}$ be vectors of the same dimension. Then $[\vec{u}]^\perp, [\vec{v}]$ is derivable if and only if $\vec{u} = \vec{v}$.*

**Proof sketch.** The "if" direction is trivial. Let us consider the "only if" direction. Assume that $\vec{u} = (n_{k-d+1}, \ldots, n_k)$, $\vec{v} = (m_{k-d+1}, \ldots, m_k)$ and $[\vec{u}]^\perp, [\vec{v}]$ is derivable. Let $\pi$ be a cut-free proof of this sequent. Since $\pi$ is a MALL proof we can apply the soundness of the sequent calculus wrt. phase semantics [28]. Consider for instance the phase space[6] $((\mathbb{Z}, +, 0), \{0\})$ and for each propositional variable $a_i$, $1 \leq i \leq k$, consider the valuation $\phi_i$ such that $\phi_i(a_i) = \{1\}$ and $\phi_i(b) = \{0\}$ for $b \neq a_i$. Soundness ensures that $[\vec{u}]^\perp, [\vec{v}]$ is valid in every phase model, that is $[\![ [\vec{u}]^\perp, [\vec{v}] ]\!]^{\phi_i} = m_i - n_i = 0$ for $k-d+1 \leq i \leq k$, that is $\vec{u} = \vec{v}$. ◀

The following technical lemma will allow us to reason by induction on the dimension via the encoding at the provability level, which is crucial to prove our forthcoming theorem.

▶ **Lemma 25.** *Let $1 \leq i \leq k$, $m \geq 0$ and $s$ an integer such that $s + m \geq 0$. Let $\vec{q}$ be an integer vector of dimension $k - i$. If $[\vec{q}], \Gamma, a_i^{s+m}, a_{i+1}^{m_{i+1}}, \ldots, a_k^{m_k}$ is provable, then so is $[\vec{r}], \Gamma, a_i^m, a_{i+1}^{m_{i+1}}, \ldots, a_k^{m_k}$ where $\vec{r} = (s, \vec{q})$ and $\vec{v} = (m, \vec{u})$.*

**Proof.** Let $\pi$ be a proof of $[\vec{q}]^\perp, \Gamma, [\vec{u}], a_i^m$. We have three cases depending on $s$: when $s$ is positive, negative, or zero.

**Case 1:** If $s$ is positive.

$$\dfrac{\dfrac{\dfrac{}{\{\, a_i^\perp, a_i \,\}_s}\,\mathsf{id} \qquad \dfrac{\pi}{[\vec{q}]^\perp, \Gamma, a_i^m, [\vec{u}]}}{[\vec{r}]^\perp, \Gamma, a_i^{s+m}, [\vec{u}]}\,\otimes^s}{[\vec{r}]^\perp, \Gamma, [\vec{v}]}\,\parr^{s+m}$$

**Case 2:** If $s$ is negative.

$$\dfrac{\dfrac{\dfrac{\pi}{[\vec{q}]^\perp, \Gamma, [\vec{u}], a_i^m}}{a_i^{|s|}, [\vec{q}]^\perp, \Gamma, [\vec{v}]}\,\parr^{s+m}}{[\vec{r}]^\perp, \Gamma, [\vec{v}]}\,\parr^{|s|}si$$

**Case 3:** If $s$ is zero, then by the encoding it is simply ignored, hence this case is trivial. ◀

We can now define the encoding of an AVASS. Fix an AVASS $\mathcal{B}$ with $|Q_\ell \times \mathsf{A}| = \alpha$, $|T_u| = \beta$, and $|T_f| = \gamma$.

- For a unary rule $t \in T_u$ of the form $(p, \vec{r}, q)$ we have $[t] \triangleq p \parr (q^\perp \otimes [\vec{r}])$.
- For a fork rule $t \in T_f$ of the form $(p, q_1, q_2)$ we have that $[t] \triangleq (p \parr ({q_1}^\perp \otimes z)) \oplus (p \parr ({q_2}^\perp \otimes z))$.
- For a "final" configuration $(q, \vec{v}) \in Q_\ell \times \mathsf{A}$, we have that $[(q, \vec{v})] \triangleq q \parr [\vec{v}]$.

$\mathcal{B}$ is encoded as $B \triangleq \mu X. \big(\mathsf{CH}_{Q_\ell \times \mathsf{A}} \oplus (\mathsf{CH}_{T_u} \parr (z \otimes X)) \oplus (\mathsf{CH}_{T_f} \parr (z \otimes X)\big)$.

[6] The *facts* of this phase space are the singletons, $\mathbb{Z}$ and $\emptyset$. One has $[\![A^\perp \otimes B^\perp]\!]^\phi = [\![A]\!]^\phi + [\![B]\!]^\phi$, $[\![A \parr B]\!]^\phi = [\![A \otimes B]\!]^{\phi\,\perp}$ when $+$ is lifted to sets of integers. In particular, the interpretations of $\otimes$ and $\parr$ coincide on formulas interpreted with singleton facts.

▶ **Theorem 26.** *The configuration $(q, \vec{v})$ is reachable in $\mathcal{B}$ iff $B, q, a_1^{v_1}, \dots, a_k^{v_k}, z$ is provable in $\mu\mathsf{MALL}^*$ where $\vec{v} = (v_1, \dots, v_k)$.*

**Proof sketch.** The idea is quite similar to the proof of Lemma 17 *i.e.* given the run tree of $(q, \vec{v})$, we produce a proof tree of the sequent $B, q, a_1^{v_1}, \dots, a_k^{v_k}, z$ that closely resembles the shape of the run tree, and vice versa. The first direction is relatively simple going through an induction on the run tree. The base case checks that leaves are valid final configurations. For the induction case, we have two cases depending on whether a unary or a fork rule has been applied at the node in question. We exhibit the proof in the case of a unary rule which exploits Lemma 25.

Suppose the unary rule $(p, \vec{r}, q)$ is applied to the node labelled $(p, \vec{v})$ where $\vec{r} = (r_1, \dots, r_k)$.

$$\dfrac{\dfrac{}{p^\perp, p}\,\mathsf{id} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{z^\perp, z}\,\mathsf{id} \quad \dfrac{\text{IH}}{q, B, a_1^{v_1+r_1}, \dots, a_k^{v_k+r_k}, z}}{q, z^\perp, z \otimes B, a_1^{v_1+r_1}, \dots, a_k^{v_k+r_k}, z}\,\otimes}{q, [\vec{r}], z \otimes B, a_1^{v_1}, \dots, a_k^{v_k}, z}\,\text{Lemma 25, } k \text{ times}}{q \parr [\vec{r}], z \otimes B, a_1^{v_1}, \dots, a_k^{v_k}, z}\,\parr}{\dfrac{p^\perp \otimes (q \parr [\vec{r}]), z \otimes B, p, a_1^{v_1}, \dots, a_k^{v_k}, z}{\dfrac{\mathsf{CH}_{T_u}, z \otimes B, q, a_1^{v_1}, \dots, a_k^{v_k}, z}{\dfrac{\mathsf{CH}_{T_u} \parr (z \otimes B), q, a_1^{v_1}, \dots, a_k^{v_k}, z}{\dfrac{\mathsf{CH}_{Q_\ell \times \mathsf{A}} \oplus (\mathsf{CH}_{T_u} \parr (z \otimes B)) \oplus (\mathsf{CH}_{T_f} \parr (z \otimes B)), q, a_1^{v_1}, \dots, a_k^{v_k}, z}{B, q, a_1^{v_1}, \dots, a_k^{v_k}, z}\,\mu}\,\oplus_2}\,\parr}\,\oplus}\,\otimes$$

where the $\beta$-ary $\oplus$ chooses the (encoding of the) rule $(p, \vec{r}, q)$.

The other direction (*i.e.* given a proof tree produce a run tree) is more involved but as in the proof of Lemma 17 exploits the stringent structure of focused proofs *i.e.* *wlog* we assume that we are given a cut-free focused proof of sequent $B, q, a_1^{v_1}, \dots, a_k^{v_k}$. We will induct on the height of the proof. It must have the following prefix.

$$\dfrac{\mathsf{CH}_{Q_\ell \times \mathsf{A}} \oplus (\mathsf{CH}_{T_u} \parr (z \otimes B)) \oplus (\mathsf{CH}_{T_f} \parr (z \otimes B)), q, a_1^{v_1}, \dots, a_k^{v_k}, z}{B, q, a_1^{v_1}, \dots, a_k^{v_k}, z}\,\mu$$

Assume that $q \in Q$ and $\{a_1, \dots, a_k\}$ are negative atoms and $z$ is a positive atom. There are now three cases depending on whether the next rule is $\oplus_1, \oplus_2$ or $\oplus_3$. Based on what is chosen, we know whether the first rule of the run tree is an axiom, or a unary rule, or a binary rule. In fact, the focusing constraint forced one to choose the particular rule as well. We will exemplify the reasoning by exhibiting the situation for the fork rule (*i.e.* $\oplus_3$). The following rule is necessarily $\parr$. Now there are two possibilities, either $\mathsf{CH}_{T_f}$ is the focus or $z \otimes B$ is the focus. Suppose the latter happens. Then, the left premisse of the tensor rule with principal formula $z \otimes B$ is of the form $z, \Gamma$. As this is a positive sequent, it must be the conclusion of an $\mathsf{id}$ rule which is not possible as $z^\perp \notin \Gamma$ and therefore $\mathsf{CH}_{T_f}$ is the focus.

Assume that the proof chooses the (encoding of the) fork rule $t = (q', q_1, q_2) \in T_f$ (by applying the correct version of the $\gamma$-ary $\oplus$). The premisse is now negative and the next rule must be a $\&$. Therefore, we have two sequents of the form $q'^\perp \otimes (q_i \parr z^\perp), z \otimes B, q, a_1^{v_1}, \dots, a_k^{v_k}, z$ for $i \in \{1, 2\}$. Let us discuss about the case when $i = 1$. (The other case is symmetric.)

As before, $z \otimes B$ cannot be the focus, so, $q'^\perp \otimes (q_1 \parr z^\perp)$ is the focus and the next rule is a $\otimes$. Since $q'^\perp$ is positive, the left premisse must be the conclusion of an $\mathsf{id}$ rule which forces $q' = q$. The right premisse is $q_1 \parr z^\perp, z \otimes B, a_1^{v_1}, \dots, a_k^{v_k}, z$ which is negative and after a $\parr$ rule, we focus on $z \otimes B$: the left premisse of the tensor must be $z, z^\perp$ as it is the conclusion

of a id rule, which leaves us with a subproof on which we can apply the induction hypothesis and we get a run-tree rooted at $(q_1, \vec{v})$. Similarly we get a run-tree rooted at $(q_2, \vec{v})$. Now we have a run tree rooted at $(q, \vec{v})$ where the first rule is the fork rule $(q, q_1, q_2)$. ◀

From Theorem 21 and Theorem 26, we have the following corollaries.

▶ **Corollary 27.** $\mu\mathsf{MALL}^*$ *is* $\Sigma^0_1$*-complete.*

▶ **Corollary 28.** $\mu\mathsf{MALL}^{\mathsf{ind}}$ *and* $\mu\mathsf{MALL}^{\omega}$ *are* $\Sigma^0_1$*-complete.*

**Proof sketch.** $\Sigma^0_1$-memership is immediate, since both $\mu\mathsf{MALL}^{\mathsf{ind}}$ and $\mu\mathsf{MALL}^{\omega}$ are systems of finitely presentable proofs that are recursively checkable.

For hardness, note that $\mu\mathsf{MALL}^{\infty} \supseteq \mu\mathsf{MALL}^{\omega} \supseteq \mu\mathsf{MALL}^{\mathsf{ind}} \supseteq \mu\mathsf{MALL}^*$ satisfies cut-elimination [4]. Since any $\mu\mathsf{MALL}^{\infty}$ proof of a $\mu$-only sequent is necessarily a finite tree, all these systems are actually conservative over $\mu\mathsf{MALL}^*$, and thus $\Sigma^0_1$-hardness is inherited. ◀

It is folklore that if $\phi(X)$ is an LK formula with a free variable $X$ then $\phi(X)$ and $\phi(\phi(\phi(X)))$ are equivalent. This immediately gives us a conservative embedding of $\mu$LK (note that this is different from $\mu$-calculus since there are no modalities) in LK with a polynomial blowup. In the same vein, [27, 26] shows that there is a conservative embedding of $\mu$LJ in LJ with an exponential blowup. MALL is known to be PSpace-complete [43]. Therefore we have the following corollary.

▶ **Corollary 29.** *There is no effectively computable reduction from* $\mu\mathsf{MALL}^*$ *(or* $\mu\mathsf{MALL}^{\mathsf{ind}}$*,* $\mu\mathsf{MALL}^{\omega}$*) to* MALL*.*

## 6 Conclusions and future work

In this work we classify the complexity of several systems for fixed point logics (cf. Figure 1). In particular, we proved that the non-wellfounded calculus $\mu\mathsf{MALL}^{\infty}$ is undecidable (via a reduction to the non-halting of Minsky machines) and prove strictly more theorems than $\mu\mathsf{MALL}^{\omega}$, its regular counterpart. We further proved that the finite provability for $\mu$MALL (in any of our systems) is already undecidable. Namely the problem is $\Sigma^0_1$-complete, via a reduction to reachability in alternating vector addition systems. One novelty of our reductions is that they are based on focusing and establishes an isomorphism between proof-trees and run-trees of Minsky machines and AVASSs.

Since its inception, linear logic was advertised as the logic for concurrency [29] and its relation with VASs (or, rather, Petri nets) has been significantly explored from both syntactic [23] and semantic [44] points of view. Our results are also cognate with this line of research. The main open questions that remain from this work is:

**Complexity of $\mu\mathsf{MALL}^{\infty}$.** There is a trivial upper bound for $\mu\mathsf{MALL}^{\infty}$ *viz.* $\Sigma^1_3$ in the analytical hierarchy. That leaves a huge gap between our $\Pi^0_1$ lower bound proved in Theorem 15. Discerning the exact complexity of $\mu\mathsf{MALL}^{\infty}$ therefore, amounts to closing this gap. Note that the undecidability of $\mu\mathsf{MALL}^*$ shows that established strategies [12, 49, 36, 37] of proving a $\Pi^0_1$ upper bound cannot be adapted to $\mu\mathsf{MALL}^{\infty}$.

**Induction vs cycles.** Is $\mu\mathsf{MALL}^{\mathsf{ind}}$ equivalent to $\mu\mathsf{MALL}^{\omega}$? This is a manifestation of the so-called *Brotherston-Simpson conjecture* in the setting of $\mu$MALL [8]. Roughly speaking, is induction as powerful as circular reasoning? Note that if the answer is indeed negative, then such a result cannot be obtained using techniques similar to Section 4.2 since in Section 5 we show that they have the same complexities.

We conclude by mentioning another pertinent direction for future work. In Section 3.3 we exhibited a purely multiplicative sequent $\nu X.X \parr X$ which has a circular proof only if we allow cuts (Figure 3c). It would be interesting to further develop regularisation procedures that blend ideas from both automata theory and proof theory, generalising the construction in Figure 3c. Naturally, by Theorem 18, no such procedure can be well-defined for all of $\mu\mathsf{MALL}^\infty$, but it is reasonable to ask if there is a middle ground.

## References


**1** Bahareh Afshari, Gerhard Jäger, and Graham E. Leigh. An infinitary treatment of full $\mu$-calculus. In Rosalie Iemhoff, Michael Moortgat, and Ruy de Queiroz, editors, *Logic, Language, Information, and Computation*, pages 17–34, Berlin, Heidelberg, 2019. Springer Berlin Heidelberg.

**2** David Baelde. On the proof theory of regular fixed points. In Martin Giese and Arild Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, 18th International Conference, TABLEAUX 2009, Oslo, Norway, July 6-10, 2009. Proceedings*, volume 5607 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2009. `doi:10.1007/978-3-642-02716-1_8`.

**3** David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. Bouncing threads for infinitary and circular proofs. In *LICS*, New York, NY, USA, 2022. Association for Computing Machinery. (to appear).

**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*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. URL: `http://www.dagstuhl.de/dagpub/978-3-95977-022-4`.

**5** David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2007. `doi:10.1007/978-3-540-75560-9_9`.

**6** Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter, editors. *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*. North-Holland, 2007. URL: `https://www.sciencedirect.com/bookseries/studies-in-logic-and-practical-reasoning/vol/3/suppl/C`.

**7** James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In Bernhard Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 78–92, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

**8** James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, October 2010. `doi:10.1093/logcom/exq052`.

**9** Paul Brunet. A complete axiomatisation of a fragment of language algebra. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, volume 152 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.CSL.2020.11`.

**10** J. Richard Buchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. URL: `http://www.jstor.org/stable/1994916`.

**11** S. Buss. Chapter 1: An introduction to proof theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*. Elsevier, 1998.

12 Wojciech Buszkowski. On action logic: Equational theories of action algebras. *Journal of Logic and Computation*, 17(1):199–217, October 2006. doi:10.1093/logcom/exl036.

13 Anupam Das. On the logical complexity of cyclic arithmetic. *Log. Methods Comput. Sci.*, 16(1), 2020. doi:10.23638/LMCS-16(1:1)2020.

14 Anupam Das, Abhishek De, and Alexis Saurin. Decision problems for linear logic with least and greatest fixed points (Extended version). working paper or preprint, April 2022. URL: https://hal.archives-ouvertes.fr/hal-03655651.

15 Anupam Das, Amina Doumane, and Damien Pous. Left-handed completeness for Kleene algebra, via cyclic proofs. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 271–289. EasyChair, 2018.

16 Anupam Das and Damien Pous. A cut-free cyclic proof system for Kleene algebra. In Renate A. Schmidt and Cláudia Nalon, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 261–277, Cham, 2017. Springer International Publishing.

17 Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time $\mu$-calculus. In *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006.

18 Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time $\mu$-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.

19 Abhishek De, Luc Pellissier, and Alexis Saurin. Canonical proof-objects for coinductive programming: Infinets with infinitely many cuts. In *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3479394.3479402.

20 P. de Groote, B. Guillaume, and S. Salvati. Vector addition tree automata. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 64–73, 2004. doi:10.1109/LICS.2004.1319601.

21 Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01676953.

22 Amina Doumane and Damien Pous. Completeness for identity-free kleene lattices. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CONCUR.2018.18.

23 Uffe Engberg and Glynn Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *CAAP '90*, pages 147–161, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

24 Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979. doi:10.1016/0022-0000(79)90046-1.

25 Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *CSL*, 2013.

26 Silvio Ghilardi, Maria João Gouveia, and Luigi Santocanale. Fixed-point elimination in the intuitionistic propositional calculus. *ACM Trans. Comput. Logic*, 21(1), September 2019. doi:10.1145/3359669.

27 Silvio Ghilardi, Maria João Gouveia, and Luigi Santocanale. Fixed-point elimination in the intuitionistic propositional calculus. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 126–141, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

28 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

**29** Jean-Yves Girard. Linear logic: Its syntax and semantics. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, pages 222–1. Cambridge University Press, 1995.

**30** Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. Association for Computing Machinery. `doi:10.1145/512927.512945`.

**31** Alexei Kopylov. Decidability of linear affine logic. *Inf. Comput.*, 164:173–198, January 2001. `doi:10.1006/inco.1999.2834`.

**32** D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 214–225, 1991. `doi:10.1109/LICS.1991.151646`.

**33** Dexter Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982. `doi:10.1016/0304-3975(82)90125-6`.

**34** Saul Aaron Kripke. The problem of entailment" (abstract). *The Journal of Symbolic Logic*, 24(4):312–326, 1959. URL: `http://www.jstor.org/stable/2963903`.

**35** Daniel Krob. Complete systems ofb-rational identities. *Theoretical Computer Science*, 89(2):207–343, 1991. `doi:10.1016/0304-3975(91)90395-I`.

**36** Stepan Kuznetsov. *-continuity vs. induction: Divide and conquer. In Guram Bezhanishvili, Giovanna D'Agostino, George Metcalfe, and Thomas Studer, editors, *Advances in Modal Logic 12, proceedings of the 12th conference on "Advances in Modal Logic," held in Bern, Switzerland, August 27-31, 2018*, pages 493–510. College Publications, 2018. URL: `http://www.aiml.net/volumes/volume12/Kuznetsov.pdf`.

**37** Stepan Kuznetsov. Complexity of commutative infinitary action logic. In Manuel A. Martins and Igor Sedlár, editors, *Dynamic Logic. New Trends and Applications*, pages 155–169, Cham, 2020. Springer International Publishing.

**38** Yves Lafont. The finite model property for various fragments of linear logic. *Journal of Symbolic Logic*, 62(4):1202–1208, 1997. `doi:10.2307/2275637`.

**39** Ranko Lazić and Sylvain Schmitz. Non-elementary complexities for branching vass, mell, and extensions. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2603088.2603129`.

**40** Ranko Lazić and Sylvain Schmitz. Nonelementary complexities for branching vass, mell, and extensions. *ACM Trans. Comput. Logic*, 16(3), June 2015. `doi:10.1145/2733375`.

**41** Patrick Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Standford University, 1992.

**42** Patrick Lincoln. Deciding provability of linear logic formulas. In *Proceedings of the Workshop on Advances in Linear Logic*, pages 109–122, USA, 1995. Cambridge University Press.

**43** Patrick Lincoln, John Mitchell, Scedrov Andre, and Natarajan Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56(1):239–311, 1992. `doi:10.1016/0168-0072(92)90075-B`.

**44** José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990. `doi:10.1016/0890-5401(90)90013-8`.

**45** Marvin L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., USA, 1967.

**46** J. F. Nash. Equilibrium points in $N$-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(48-49), 1950.

**47** Damian Niwiński. Fixed point characterization of infinite behavior of finite-state systems. *Theor. Comput. Sci.*, 189(1–2):1–69, December 1997. `doi:10.1016/S0304-3975(97)00039-X`.

**48** 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.

**49** Ewa Palka. An infinitary sequent system for the equational theory of *-continuous action lattices. *Fundam. Inf.*, 78(2):295–309, April 2007.

**50** Michael Oser Rabin. *Automata on Infinite Objects and Church's Problem.* American Mathematical Society, USA, 1972.

**51** 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*, pages 357–371, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

**52** HAROLD SCHELLINX. Some Syntactical Observations on Linear Logic. *Journal of Logic and Computation*, 1(4):537–559, September 1991. doi:10.1093/logcom/1.4.537.

**53** Dana Scott. Outline of a mathematical theory of computation. Technical Report PRG02, OUCL, November 1970.

**54** Alex Simpson. Cyclic arithmetic is equivalent to peano arithmetic. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 283–300, 2017. doi:10.1007/978-3-662-54458-7_17.

**55** Lutz Straßburger. On the decision problem for MELL. *Theoretical Computer Science*, 768:91–98, 2019. doi:10.1016/j.tcs.2019.02.022.

**56** Thomas Studer. On the proof theory of the modal mu-calculus. *Studia Logica: An International Journal for Symbolic Logic*, 89(3):343–363, 2008. URL: http://www.jstor.org/stable/40268983.

**57** Alasdair Urquhart. The complexity of decision procedures in relevance logic ii. *The Journal of Symbolic Logic*, 64(4):1774–1802, 1999. URL: http://www.jstor.org/stable/2586811.

**58** Igor Walukiewicz. *A complete deductive system for the $\mu$-calculus.* PhD thesis, Warsaw University, 1994.

**59** Igor Walukiewicz. Completeness of Kozen's axiomatisation of the propositional $\mu$-calculus. In *LICS 95, San Diego, California, USA, June 26-29, 1995*, pages 14–24, 1995.

# Linear Lambda-Calculus is Linear

**Alejandro Díaz-Caro**
Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes, Bernal, Buenos Aires, Argentina
Instituto de Ciencias de la Computación, CONICET / Universidad de Buenos Aires, Buenos Aires, Argentina

**Gilles Dowek**
Inria, Paris, France
ENS Paris-Saclay, France

## Abstract

We prove a linearity theorem for an extension of linear logic with addition and multiplication by a scalar: the proofs of some propositions in this logic are linear in the algebraic sense. This work is part of a wider research program that aims at defining a logic whose proof language is a quantum programming language.

**2012 ACM Subject Classification** Theory of computation → Proof theory; Theory of computation → Lambda calculus; Theory of computation → Linear logic; Theory of computation → Quantum computation theory

**Keywords and phrases** Proof theory, Lambda calculus, Linear logic, Quantum computing

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.21

**Related Version** *Full Version*: `arXiv:2201.11221`

**Funding** STIC-AmSud 21STIC10, ECOS-Sud A17C03, and the French-Argentinian IRP SINFIN.
*Alejandro Díaz-Caro*: PIP 11220200100368CO and PICT-2019-1272.

**Acknowledgements** The authors want to thank Thomas Ehrhard, Jean-Baptiste Joinet, Jean-Pierre Jouannaud, Dale Miller, Alberto Naibo, Simon Perdrix, Alex Tsokurov, and Lionel Vaux for useful discussions.

## 1 Introduction

The name of linear logic [10] suggests that this logic has some relation with the algebraic notion of linearity. A common account of this relation is that a proof of a linear implication between two propositions $A$ and $B$ should not be any function mapping proofs of $A$ to proofs of $B$, but a linear one. This idea has been fruitfully exploited to build models of linear logic (for example [3,9,11]), but it seems difficult to even formulate it within the proof language itself. Indeed, expressing the properties $f(u+v) = f(u) + f(v)$ and $f(a.u) = a.f(u)$ requires an addition and a multiplication by a scalar, that are usually not present in proof languages.

The situation has changed with quantum programming languages [1,2,4,6,8,12,14] and the algebraic $\lambda$-calculus [13], that mix some usual constructions of programming languages with algebraic operations. More specifically, several extensions of the lambda-calculus, or of a language of proof-terms, with addition and multiplication by a scalar have been proposed [2,5,13].

In this paper, we investigate an extension of linear logic with addition and multiplication by a scalar, the $\mathcal{L}^{\mathcal{S}}$-logic (where $\mathcal{S}$ denotes the field of scalars used), and we prove a linearity theorem: if $f$ is a proof of an implication between two propositions of some specific form, then $f(u+v) = f(u) + f(v)$ and $f(a.u) = a.f(u)$.

© Alejandro Díaz-Caro and Gilles Dowek;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 21; pp. 21:1–21:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This work is part of a wider research program that aims at determining in which way propositional logic must be extended or restricted, so that its proof language becomes a quantum programming language. There are two main issues in the design of a quantum programming language: the first is to take into account the linearity of the unitary operators and, for instance, avoid cloning, and the second is to express the information-erasure, non-reversibility, and non-determinism of the measurement. In [5], we addressed the question of the measurement. In this paper, we address that of linearity.

## 1.1 Interstitial rules

To extend linear logic with addition and multiplication by a scalar, we proceed, like in [5, long version], in two steps: we first add interstitial rules and then scalars.

An interstitial rule is a deduction rule whose premises are identical to its conclusion. In the $\mathcal{L}^{\mathcal{S}}$-logic, we consider two such rules

$$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A}{\Gamma \vdash A}\ \text{sum} \qquad\qquad \dfrac{\Gamma \vdash A}{\Gamma \vdash A}\ \text{prod}$$

Adding these rules permits to build proofs that cannot be reduced, because the introduction rule of some connective and its elimination rule are separated by an interstitial rule, for example

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-i} \quad \dfrac{\dfrac{\pi_3}{\Gamma \vdash A} \quad \dfrac{\pi_4}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-i}}{\Gamma \vdash A \wedge B}\ \text{sum} \quad \dfrac{\pi_5}{\Gamma, A \vdash C}}{\Gamma \vdash C}\ \wedge\text{-e1}$$

Reducing such a proof, sometimes called a commuting cut, requires reduction rules to commute the rule sum either with the elimination rule below or with the introduction rules above.

As the commutation with the introduction rules above is not always possible, for example in the proof

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\ \vee\text{-i1} \quad \dfrac{\dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \vee B}\ \vee\text{-i2}}{\Gamma \vdash A \vee B}\ \text{sum}$$

the commutation with the elimination rule below is often preferred. In this paper, we favour the commutation of the interstitial rules with the introduction rules, rather than with the elimination rules, whenever it is possible, that is for all connectives except the disjunction. For example, the proof

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-i} \quad \dfrac{\dfrac{\pi_3}{\Gamma \vdash A} \quad \dfrac{\pi_4}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-i}}{\Gamma \vdash A \wedge B}\ \text{sum}$$

reduces to

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \quad \dfrac{\pi_3}{\Gamma \vdash A}}{\Gamma \vdash A}\ \text{sum} \quad \dfrac{\dfrac{\pi_2}{\Gamma \vdash B} \quad \dfrac{\pi_4}{\Gamma \vdash B}}{\Gamma \vdash B}\ \text{sum}}{\Gamma \vdash A \wedge B}\ \wedge\text{-i}$$

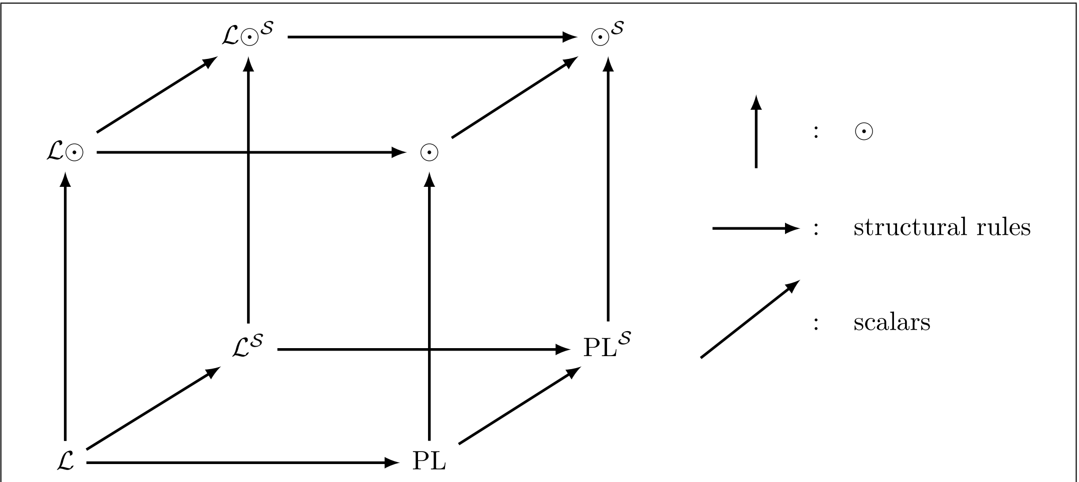



Figure 1 Eight logics.

Such a commutation yields a stronger introduction property for the considered connective.

For coherence, we commute both rules sum and prod with the elimination rule of the disjunction, rather that with its introduction rules. But, for the rule prod, both choices are possible.

## 1.2 Scalars

We then consider a field $\mathcal{S}$ of scalars and replace the introduction rule of the connective $\top$ with a family of rules $\top$-i$(a)$, one for each scalar, and the rule prod with a family of rules prod$(a)$, also one for each scalar

$$\frac{}{\Gamma \vdash \top}\ \top\text{-i}(a) \qquad\qquad \frac{\Gamma \vdash A}{\Gamma \vdash A}\ \text{prod}(a)$$

## 1.3 The connective $\odot$

Besides interstitial rules and scalars, we have introduced, in [5, long version], a new connective $\odot$ (read "sup" for "superposition"), that has an introduction rule $\odot$-i similar to that of the conjunction, two elimination rules $\odot$-e1 and $\odot$-e2 similar to those of the conjunction, but also a third elimination rule $\odot$-e similar to that of the disjunction.

The elimination rules $\odot$-e1 and $\odot$-e2 are used to express the information-preserving, reversible, and deterministic operations, such as the unitary transformations of quantum computing. The elimination rule $\odot$-e is used to express the information-erasing, non-reversible, and non-deterministic operations, such as quantum measurement. We will come back to this full system at Section 6.1 (the $\odot$ rules are listed at Figure 4).

Starting from propositional logic with the interstitial rules sum and prod, we can thus either add scalars, or the connective $\odot$, or both. This yields the four logics on the right face of the cube of Figure 1: PL is propositional logic with the interstitial rules sum and prod, $\text{PL}^{\mathcal{S}}$ is propositional logic with the interstitial rules and scalars, $\odot$ is propositional logic with the interstitial rules and the connective $\odot$, and $\odot^{\mathcal{S}}$ is propositional logic with the interstitial rules, the connective $\odot$, and scalars.

## 1.4 Linearity

The proof language of the $\odot^{\mathcal{S}}$-logic is a quantum programming language, as quantum algorithms can be expressed in it. However, this language addresses the question of quantum measurement, but not the that of linearity, and non-linear functions, such as cloning operators, can also be expressed in it.

This leads to introduce, in this paper, a linear variant of the $\odot^{\mathcal{S}}$-logic, and prove a linearity theorem for it.

More generally, we can introduce, on the left face of the cube of Figure 1, a linear variant for each of the four logics of the right face: $\mathcal{L}$ is linear logic with the interstitial rules sum and prod, $\mathcal{L}^{\mathcal{S}}$ is linear logic with the interstitial rules and scalars, $\mathcal{L}\odot$ is linear logic with the interstitial rules and the connective $\odot$, and $\mathcal{L}\odot^{\mathcal{S}}$ is linear logic with the interstitial rules, the connective $\odot$, and scalars.

Our goal is to prove a linearity theorem for the proof language of the $\mathcal{L}\odot^{\mathcal{S}}$-logic. But such a theorem does not hold for the full $\mathcal{L}\odot^{\mathcal{S}}$-logic, that contains the rule $\odot$-e, that enables to express measurement operators, which are not linear. Thus, our linearity theorem should concern the fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-logic without this rule. But, if $\odot$-e rule is excluded, the connective $\odot$ is just the conjunction, and this fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-logic is the $\mathcal{L}^{\mathcal{S}}$-logic.

So, for a greater generality, we prove our linearity theorem for the $\mathcal{L}^{\mathcal{S}}$-logic: linear logic with the interstitial rules and scalars, but without the $\odot$ connective, and discuss, at the end of the paper, how this result extends to the $\mathcal{L}\odot^{\mathcal{S}}$-logic.

## 1.5 Linear connectives

In the $\mathcal{L}^{\mathcal{S}}$-logic, we have to make a choice of connectives.

In intuitionistic linear logic, there is no multiplicative falsehood, no additive implication, and no multiplicative disjunction. Thus, we have two possible truths and two possible conjunctions, but only one possible falsehood, implication, and disjunction.

In the $\mathcal{L}^{\mathcal{S}}$-logic, we have chosen a multiplicative truth, an additive falsehood, a multiplicative implication, an additive conjunction, and an additive disjunction. The rule sum also is additive. The reasons for this choice of connectives will be justified in Remarks 2.1, 5.1, and 5.2.

These symbols are often written 1, 0, $\multimap$, &, and $\oplus$. As we use only one conjunction, one disjunction, etc., to make our paper more accessible to readers who are not familiar with linear logic (and also because we have several zeros, for scalars, vectors, etc.), we use the usual symbols $\top$, $\bot$, $\Rightarrow$, $\wedge$, and $\vee$ instead. Of course, notations are arbitrary and the notations of linear logic can also be used.

The introduction rule for the additive conjunction is the same as that in usual natural deduction

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-i}$$

In particular, the proofs of $A$ and $B$ are in the same context $\Gamma$. But, in the elimination rule

$$\frac{\Gamma \vdash A \wedge B \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}\ \wedge\text{-e1}$$

the proof of $A \wedge B$ and that of $C$ must be in contexts $\Gamma$ and $\Delta, A$.

$$\frac{}{x:A \vdash x:A}\text{ ax} \qquad \frac{\Gamma \vdash t:A \quad \Gamma \vdash u:A}{\Gamma \vdash t \boldsymbol{+} u:A}\text{ sum} \qquad \frac{\Gamma \vdash t:A}{\Gamma \vdash a \bullet t:A}\text{ prod}(a)$$

$$\frac{}{\vdash a.\star:\top}\ \top\text{-i}(a) \qquad \frac{\Gamma \vdash t:\top \quad \Delta \vdash u:A}{\Gamma,\Delta \vdash \delta_\top(t,u):A}\ \top\text{-e} \qquad \frac{\Gamma \vdash t:\bot}{\Gamma,\Delta \vdash \delta_\bot(t):C}\ \bot\text{-e}$$

$$\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x.t:A \Rightarrow B} \Rightarrow\text{-i} \qquad \frac{\Gamma \vdash t:A \Rightarrow B \quad \Delta \vdash u:A}{\Gamma,\Delta \vdash t\ u:B} \Rightarrow\text{-e}$$

$$\frac{\Gamma \vdash t:A \quad \Gamma \vdash u:B}{\Gamma \vdash \langle t,u\rangle:A \wedge B}\ \wedge\text{-i}$$

$$\frac{\Gamma \vdash t:A \wedge B \quad \Delta, x:A \vdash u:C}{\Gamma,\Delta \vdash \delta^1_\wedge(t,x.u):C}\ \wedge\text{-e1} \qquad \frac{\Gamma \vdash t:A \wedge B \quad \Delta, x:B \vdash u:C}{\Gamma,\Delta \vdash \delta^2_\wedge(t,x.u):C}\ \wedge\text{-e2}$$

$$\frac{\Gamma \vdash t:A}{\Gamma \vdash inl(t):A \vee B}\ \vee\text{-i1} \qquad \frac{\Gamma \vdash t:B}{\Gamma \vdash inr(t):A \vee B}\ \vee\text{-i2}$$

$$\frac{\Gamma \vdash t:A \vee B \quad \Delta, x:A \vdash u:C \quad \Delta, y:B \vdash v:C}{\Gamma,\Delta \vdash \delta_\vee(t,x.u,y.v):C}\ \vee\text{-e}$$

**Figure 2** The deduction rules of the $\mathcal{L}^\mathcal{S}$-calculus.

In this paper, we first define the $\mathcal{L}^\mathcal{S}$-logic and its proof-language: the $\mathcal{L}^\mathcal{S}$-calculus, and prove that it verifies the subject reduction, confluence, termination, and introduction properties (Section 2). We then show how the vectors of $\mathcal{S}^n$ can be expressed in this calculus and how the irreducible closed proofs of some propositions are equipped with a structure of vector space (Section 3). We prove that all linear functions from $\mathcal{S}^m$ to $\mathcal{S}^n$ can be expressed as proofs of an implication between such propositions (Section 4). We then prove the main result of this paper: that, conversely, all the proofs of implications between such propositions are linear (Section 5). Finally, we show how this result extents to the proof language of the $\mathcal{L}\odot^\mathcal{S}$-logic and how this language is a quantum programming language (Section 6).

Most proofs are omitted from this conference paper, they can be found in its long version.

## 2 The $\mathcal{L}^\mathcal{S}$-calculus

Let $\mathcal{S}$ be a field of *scalars*, for instance $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$.

The propositions of the $\mathcal{L}^\mathcal{S}$-logic are those of propositional logic

$$A = \top \mid \bot \mid A \Rightarrow A \mid A \wedge A \mid A \vee A$$

The proof-terms of this logic are

$$\begin{aligned} t = {} & x \mid t \boldsymbol{+} u \mid a \bullet t \mid a.\star \mid \delta_\top(t,u) \mid \delta_\bot(t) \\ & \mid \lambda x.t \mid (t\ u) \mid \langle t,u\rangle \mid \delta^1_\wedge(t,x.u) \mid \delta^2_\wedge(t,x.u) \\ & \mid inl(t) \mid inr(t) \mid \delta_\vee(t,x.u,y.v) \end{aligned}$$

where $a$ is a scalar.

The variables $x$ express the proofs built with the rule axiom, the terms $t \boldsymbol{+} u$ those built with the rule sum, the terms $a \bullet t$ those built with the family of rules prod($a$), the terms $a.\star$ those built with the family of rules $\top$-i($a$), the terms $\delta_\top(t,u)$ those built with the rule $\top$-e, the terms $\delta_\bot(t)$ those built with the rule $\bot$-e, the terms $\lambda x.t$ those built with the rule $\Rightarrow$-i, the terms $t\ u$ those built with the rule $\Rightarrow$-e, the terms $\langle t,u\rangle$ those built with the rule $\wedge$-i,

$$\begin{aligned}
\delta_\top(a.\star, t) &\longrightarrow a \bullet t\\
(\lambda x.t)\ u &\longrightarrow (u/x)t\\
\delta^1_\wedge(\langle t, u\rangle, x.v) &\longrightarrow (t/x)v\\
\delta^2_\wedge(\langle t, u\rangle, x.v) &\longrightarrow (u/x)v\\
\delta_\vee(\mathit{inl}(t), x.v, y.w) &\longrightarrow (t/x)v\\
\delta_\vee(\mathit{inr}(u), x.v, y.w) &\longrightarrow (u/y)w\\
\\
a.\star \boldsymbol{+} b.\star &\longrightarrow (a+b).\star\\
(\lambda x.t) \boldsymbol{+} (\lambda x.u) &\longrightarrow \lambda x.(t \boldsymbol{+} u)\\
\langle t, u\rangle \boldsymbol{+} \langle v, w\rangle &\longrightarrow \langle t \boldsymbol{+} v, u \boldsymbol{+} w\rangle\\
\delta_\vee(t \boldsymbol{+} u, x.v, y.w) &\longrightarrow \delta_\vee(t, x.v, y.w) \boldsymbol{+} \delta_\vee(u, x.v, y.w)\\
\\
a \bullet b.\star &\longrightarrow (a \times b).\star\\
a \bullet \lambda x.t &\longrightarrow \lambda x.a \bullet t\\
a \bullet \langle t, u\rangle &\longrightarrow \langle a \bullet t, a \bullet u\rangle\\
\delta_\vee(a \bullet t, x.v, y.w) &\longrightarrow a \bullet \delta_\vee(t, x.v, y.w)
\end{aligned}$$

**Figure 3** The reduction rules of the $\mathcal{L}^{\mathcal{S}}$-calculus.

the terms $\delta^1_\wedge(t, x.u)$ and $\delta^2_\wedge(t, x.u)$ those built with the rules $\wedge$-e1 and $\wedge$-e2, the terms $\mathit{inl}(t)$ and $\mathit{inr}(t)$ those built with the rules $\vee$-i1 and $\vee$-i2, and the terms $\delta_\vee(t, x.u, y.v)$ those built with the rule $\vee$-e.

The proofs of the form $\star$, $\lambda x.t$, $\langle t, u\rangle$, $\mathit{inl}(t)$, and $\mathit{inr}(t)$ are called *introductions*, and those of the form $\delta_\top(t, u)$, $\delta_\bot(t)$, $t\ u$, $\delta^1_\wedge(t, x.u)$, $\delta^2_\wedge(t, x.u)$, and $\delta_\vee(t, x.u, y.v)$ *eliminations*. The variables and the proofs of the form $t \boldsymbol{+} u$ and $a \bullet t$ are neither introductions nor eliminations.

The $\alpha$-equivalence relation and the free and bound variables of a proof-term are defined as usual. Proof-terms are defined modulo $\alpha$-equivalence. A proof-term is closed if it contains no free variables. We write $(u/x)t$ for the substitution of $u$ for $x$ in $t$ and if $FV(t) \subseteq \{x\}$, we also use the notation $t\{u\}$.

The typing rules are those of Figure 2. These typing rules are exactly deduction rules of linear natural deduction for the multiplicative truth, the additive falsehood, the multiplicative implication, the additive conjunction, and the additive disjunction, with proof-terms, with two differences: the interstitial rules and the scalars.

The reduction rules are those of Figure 3. As usual, the reduction relation is written $\longrightarrow$, its inverse $\longleftarrow$, its reflexive-transitive closure $\longrightarrow^*$, the reflexive-transitive closure of its inverse ${}^*\!\longleftarrow$, and its reflexive-symmetric-transitive closure $\equiv$. The first six rules correspond to the reduction of cuts on the connectives $\top$, $\Rightarrow$, $\wedge$, and $\vee$. The eight others enable to commute the interstitial rules sum and prod with the introduction rules of the connectives $\top$, $\Rightarrow$, and $\wedge$, and with the elimination rule of the connective $\vee$. For instance, the rule

$$\langle t, u\rangle \boldsymbol{+} \langle v, w\rangle \longrightarrow \langle t \boldsymbol{+} v, u \boldsymbol{+} w\rangle$$

pushes the symbol $\boldsymbol{+}$ inside the pair. In a calculus without scalars we would have the zero-ary commutation rules

$$\star \boldsymbol{+} \star \longrightarrow \star \qquad\qquad \bullet \star \longrightarrow \star$$

In the rules with scalars the scalars are added in the first case and multiplied in the second

$$a.\star \boldsymbol{+} b.\star \longrightarrow (a+b).\star \qquad\qquad a \bullet b.\star \longrightarrow (a \times b).\star$$

▶ Remark 2.1. The rule $\langle t, u\rangle + \langle v, w\rangle \longrightarrow \langle t + v, u + w\rangle$ is possible because the conjunction is additive. If it were multiplicative, from $\Gamma \vdash \langle t, u\rangle : A \wedge B$ and $\Gamma \vdash \langle v, w\rangle : A \wedge B$, we could deduce that there exist $\Gamma_1$, $\Gamma_2$, $\Gamma'_1$, $\Gamma'_2$ such that $\Gamma_1 \vdash t : A$, $\Gamma_2 \vdash u : B$, $\Gamma'_1 \vdash v : A$, $\Gamma'_2 \vdash w : B$, and $\Gamma_1, \Gamma_2 = \Gamma'_1, \Gamma'_2 = \Gamma$, but $\Gamma_1$ and $\Gamma'_1$ could be different, and we would not be able to type $t + v$. This is the justification for the choice of the additive disjunction in the $\mathcal{L}^{\mathcal{S}}$-calculus and the exclusion of the multiplicative one. This remark is also key in the subject reduction proof below.

The $\mathcal{L}^{\mathcal{S}}$-calculus has the subject reduction, confluence, termination, and introduction properties. The subject reduction property is non trivial as shown by the remark above, but its proof uses standard methods. The termination property and the introduction properties are consequences of the termination and of the introduction properties of the $\odot^{\mathcal{S}}$-calculus. The full proofs are given in the long version of the paper.

▶ **Theorem 2.2** (Subject reduction). *If $\Gamma \vdash t : A$ and $t \longrightarrow u$, then $\Gamma \vdash u : A$.*

▶ **Theorem 2.3** (Confluence). *The $\mathcal{L}^{\mathcal{S}}$-calculus is confluent.*

▶ **Theorem 2.4** (Termination). *The $\mathcal{L}^{\mathcal{S}}$-calculus is strongly terminating.*

▶ **Theorem 2.5** (Introduction). *Let $t$ be a closed irreducible proof of $A$.*
- *If $A$ has the form $\top$, then $t$ has the form $a.\star$.*
- *The proposition $A$ is not $\bot$.*
- *If $A$ has the form $B \Rightarrow C$, then $t$ has the form $\lambda x.u$.*
- *If $A$ has the form $B \wedge C$, then $t$ has the form $\langle u, v\rangle$.*
- *If $A$ has the form $B \vee C$, then $t$ has the form $inl(u)$, $inr(u)$, $u + v$, or $a \bullet u$.*

## 3 Vectors

As there is one rule $\top$-i for each scalar $a$, there is one closed irreducible proof $a.\star$ for each scalar $a$. Thus, the closed irreducible proofs $a.\star$ of $\top$ are in one-to-one correspondence with the elements of $\mathcal{S}$. Therefore, the proofs $\langle a.\star, b.\star\rangle$ of $\top \wedge \top$ are in one-to-one with the elements of $\mathcal{S}^2$, the proofs $\langle\langle a.\star, b.\star\rangle, c.\star\rangle$ of $(\top \wedge \top) \wedge \top$, and also the proofs $\langle a.\star, \langle b.\star, c.\star\rangle\rangle$ of $\top \wedge (\top \wedge \top)$, are in one-to-one correspondence with the elements of $\mathcal{S}^3$, etc.

Thus, as any vector space of finite dimension $n$ is isomorphic to $\mathcal{S}^n$, we have a way to express the vectors of any $\mathcal{S}$-vector space of finite dimension. Yet, choosing an isomorphism between a vector space and $\mathcal{S}^n$ amounts to choosing a basis in this vector space, thus the expression of a vector depends on the choice of a basis. This situation is analogous to that of matrix formalisms. Matrices can represent vectors and linear functions, but the matrix representation is restricted to finite dimensional vector spaces, and the representation of a vector depends on the choice of a basis. A change of basis in the vector space is reflected by the use of a transformation matrix.

▶ **Definition 3.1** (The set $\mathcal{V}$). *The set $\mathcal{V}$ is inductively defined as follows: $\top \in \mathcal{V}$, and if $A$ and $B$ are in $\mathcal{V}$, then so is $A \wedge B$.*

We now show that if $A \in \mathcal{V}$, then the set of closed irreducible proofs of $A$ has a vector space structure.

▶ **Definition 3.2** (Zero vector). *If $A \in \mathcal{V}$, we define the proof $0_A$ of $A$ by induction on $A$. If $A = \top$, then $0_A = 0.\star$. If $A = A_1 \wedge A_2$, then $0_A = \langle 0_{A_1}, 0_{A_2}\rangle$.*

▶ **Definition 3.3** (Additive inverse). *If $A \in \mathcal{V}$, and $t$ is a proof of $A$, we define the proof $-t$ of $A$ by induction on $A$. If $A = \top$, then $t$ reduces to $a.\star$, we let $-t = (-a).\star$. If $A = A_1 \wedge A_2$, $t$ reduces to $\langle t_1, t_2 \rangle$ where $t_1$ is a proof of $A_1$ and $t_2$ of $A_1$. We let $-t = \langle -t_1, -t_2 \rangle$.*

▶ **Lemma 3.4.** *If $A \in \mathcal{V}$ and $t$, $t_1$, $t_2$, and $t_3$ are closed proofs of $A$, then*

1. $(t_1 \mathbin{\boldsymbol{+}} t_2) \mathbin{\boldsymbol{+}} t_3 \equiv t_1 \mathbin{\boldsymbol{+}} (t_2 \mathbin{\boldsymbol{+}} t_3)$
2. $t_1 \mathbin{\boldsymbol{+}} t_2 \equiv t_2 \mathbin{\boldsymbol{+}} t_1$
3. $t \mathbin{\boldsymbol{+}} 0_A \equiv t$
4. $t \mathbin{\boldsymbol{+}} -t \equiv 0_A$
5. $a \bullet b \bullet t \equiv (a \times b) \bullet t$
6. $1 \bullet t \equiv t$
7. $a \bullet (t_1 \mathbin{\boldsymbol{+}} t_2) \equiv a \bullet t_1 \mathbin{\boldsymbol{+}} a \bullet t_2$
8. $(a + b) \bullet t \equiv a \bullet t \mathbin{\boldsymbol{+}} b \bullet t$

▶ **Definition 3.5** (Dimension of a propositon in $\mathcal{V}$). *To each proposition $A \in \mathcal{V}$, we associate a positive natural number $d(A)$, which is the number of occurrences of the symbol $\top$ in $A$: $d(\top) = 1$ and $d(B \wedge C) = d(B) + d(C)$.*

If $A \in \mathcal{V}$ and $d(A) = n$, then the closed normal proofs of $A$ and the vectors of $\mathcal{S}^n$ are in one-to-one correspondence: to each closed irreducible proof $t$ of $A$, we associate a vector $\underline{t}$ of $\mathcal{S}^n$ and to each vector $\mathbf{u}$ of $\mathcal{S}^n$, we associate a closed irreducible proof $\overline{\mathbf{u}}^A$ of $A$.

▶ **Definition 3.6** (One-to-one correspondance). *Let $A \in \mathcal{V}$ with $d(A) = n$. To each closed irreducible proof $t$ of $A$, we associate a vector $\underline{t}$ of $\mathcal{S}^n$ as follows.*

- *If $A = \top$, then $t = a.\star$. We let $\underline{t} = (\,a\,)$.*
- *If $A = A_1 \wedge A_2$, then $t = \langle u, v \rangle$. We let $\underline{t}$ be the vector with two blocks $\underline{u}$ and $\underline{v}$: $\underline{t} = \left(\begin{smallmatrix}\underline{u}\\ \underline{v}\end{smallmatrix}\right)$.*

*To each vector $\mathbf{u}$ of $\mathcal{S}^n$, we associate a closed irreducible proof $\overline{\mathbf{u}}^A$ of $A$.*

- *If $n = 1$, then $\mathbf{u} = (\,a\,)$. We let $\overline{\mathbf{u}}^A = a.\star$.*
- *If $n > 1$, then $A = A_1 \wedge A_2$, let $n_1$ and $n_2$ be the dimensions of $A_1$ and $A_2$. Let $\mathbf{u}_1$ and $\mathbf{u}_2$ be the two blocks of $\mathbf{u}$ of $n_1$ and $n_2$ lines, so $\mathbf{u} = \left(\begin{smallmatrix}\mathbf{u}_1\\ \mathbf{u}_2\end{smallmatrix}\right)$. We let $\overline{\mathbf{u}}^A = \langle \overline{\mathbf{u}_1}^{A_1}, \overline{\mathbf{u}_2}^{A_2} \rangle$.*

We extend the definition of $\underline{t}$ to any closed proof of $A$, $\underline{t}$ is by definition $\underline{t'}$ where $t'$ is the irreducible form of $t$.

The next lemmas show that the symbol $\boldsymbol{+}$ expresses the sum of vectors and the symbol $\bullet$, the product of a vector by a scalar.

▶ **Lemma 3.7** (Sum of two vectors). *Let $A \in \mathcal{V}$, and $u$ and $v$ be two closed proofs of $A$. Then, $\underline{u \mathbin{\boldsymbol{+}} v} = \underline{u} + \underline{v}$.*

▶ **Lemma 3.8** (Product of a vector by a scalar). *Let $A \in \mathcal{V}$ and $u$ be a closed proof of $A$. Then $\underline{a \bullet u} = a\underline{u}$.*

▶ Remark 3.9. We have seen that the rules

$$a.\star \mathbin{\boldsymbol{+}} b.\star \longrightarrow (a+b).\star \qquad a \bullet b.\star \longrightarrow (a \times b).\star$$
$$\langle t, u \rangle \mathbin{\boldsymbol{+}} \langle v, w \rangle \longrightarrow \langle t \mathbin{\boldsymbol{+}} v, u \mathbin{\boldsymbol{+}} w \rangle \qquad a \bullet \langle t, u \rangle \longrightarrow \langle a \bullet t, a \bullet u \rangle$$

come from the rules of a calculus without scalars

$$\star \mathbin{\boldsymbol{+}} \star \longrightarrow \star \qquad \bullet\star \longrightarrow \star$$
$$\langle t, u \rangle \mathbin{\boldsymbol{+}} \langle v, w \rangle \longrightarrow \langle t \mathbin{\boldsymbol{+}} v, u \mathbin{\boldsymbol{+}} w \rangle \qquad \bullet\langle t, u \rangle \longrightarrow \langle \bullet t, \bullet u \rangle$$

that are commutation rules between the interstitial rules, sum and prod, and introduction rules $\top$-i and $\wedge$-i.

Now, these rules appear to be also vector calculation rules.

## 4 Matrices

We now want to prove that if $A, B \in \mathcal{V}$ with $d(A) = m$ and $d(B) = n$, and $F$ is a linear function from $\mathcal{S}^m$ to $\mathcal{S}^n$, then there exists a closed proof $f$ of $A \Rightarrow B$ such that, for all vectors $\mathbf{u} \in \mathcal{S}^m$, $\underline{f\ \overline{\mathbf{u}}^A} = F(\mathbf{u})$. This can equivalently be formulated as the fact that if $M$ is a matrix with $m$ columns and $n$ lines, then there exists a closed proof $f$ of $A \Rightarrow B$ such that for all vectors $\mathbf{u} \in \mathcal{S}^m$, $\underline{f\ \overline{\mathbf{u}}^A} = M\mathbf{u}$.

This theorem has been proved for the $\odot^{\mathcal{S}}$-calculus in [5, long version]. The proof of the following theorem is just a check that the construction given there verifies the linearity constraints of the $\mathcal{L}^{\mathcal{S}}$-calculus.

▶ **Theorem 4.1** (Matrices). *Let $A, B \in \mathcal{V}$ with $d(A) = m$ and $d(B) = n$ and let $M$ be a matrix with $m$ columns and $n$ lines, then there exists a closed proof $t$ of $A \Rightarrow B$ such that, for all vectors $\mathbf{u} \in \mathcal{S}^m$, $\underline{t\ \overline{\mathbf{u}}^A} = M\mathbf{u}$.*

**Proof.** By induction on $A$.

- If $A = \top$, then $M$ is a matrix of one column and $n$ lines. Hence, it is also a vector of $n$ lines. We take

  $$t = \lambda x.\delta_\top(x, \overline{M}^B)$$

  Let $\mathbf{u} \in \mathcal{S}^1$, $\mathbf{u}$ has the form $\begin{pmatrix} a \end{pmatrix}$ and $\overline{\mathbf{u}}^A = a.\star$.
  Then, using Lemma 3.8, we have $\underline{t\ \overline{\mathbf{u}}^A} = \underline{\delta_\top(\overline{\mathbf{u}}^A, \overline{M}^B)} = \underline{\delta_\top(a.\star, \overline{M}^B)} = \underline{a \bullet \overline{M}^B} = a\underline{\overline{M}^B} = aM = M\begin{pmatrix} a \end{pmatrix} = M\mathbf{u}$.
- If $A = A_1 \wedge A_2$, then let $d(A_1) = m_1$ and $d(A_2) = m_2$. Let $M_1$ and $M_2$ be the two blocks of $M$ of $m_1$ and $m_2$ columns, so $M = \begin{pmatrix} M_1 & M_2 \end{pmatrix}$.
  By induction hypothesis, there exist closed proofs $t_1$ and $t_2$ of the propositions $A_1 \Rightarrow B$ and $A_2 \Rightarrow B$ such that, for all vectors $\mathbf{u}_1 \in \mathcal{S}^{m_1}$ and $\mathbf{u}_2 \in \mathcal{S}^{m_2}$, we have $\underline{t_1\ \overline{\mathbf{u}_1}^{A_1}} = M_1\mathbf{u}_1$ and $\underline{t_2\ \overline{\mathbf{u}_2}^{A_2}} = M_2\mathbf{u}_2$. We take

  $$t = \lambda x.(\delta^1_\wedge(x, y.(t_1\ y)) \boldsymbol{+} \delta^2_\wedge(x, z.(t_2\ z)))$$

  Let $\mathbf{u} \in \mathcal{S}^m$, and $\mathbf{u}_1$ and $\mathbf{u}_2$ be the two blocks of $m_1$ and $m_2$ lines of $\mathbf{u}$, so $\mathbf{u} = \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}$, and $\overline{\mathbf{u}}^A = \langle \overline{\mathbf{u}_1}^{A_1}, \overline{\mathbf{u}_2}^{A_2} \rangle$.
  Then, using Lemma 3.7, $\underline{t\ \overline{\mathbf{u}}^A} = \underline{\delta^1_\wedge(\langle \overline{\mathbf{u}_1}^{A_1}, \overline{\mathbf{u}_2}^{A_2} \rangle, y.(t_1\ y)) \boldsymbol{+} \delta^2_\wedge(\langle \overline{\mathbf{u}_1}^{A_1}, \overline{\mathbf{u}_2}^{A_2} \rangle, z.(t_2\ z))} = \underline{(t_1\ \overline{\mathbf{u}_1}^{A_1}) \boldsymbol{+} (t_2\ \overline{\mathbf{u}_2}^{A_2})} = \underline{t_1\ \overline{\mathbf{u}_1}^{A_1}} + \underline{t_2\ \overline{\mathbf{u}_2}^{A_2}} = M_1\mathbf{u}_1 + M_2\mathbf{u}_2 = \begin{pmatrix} M_1 & M_2 \end{pmatrix}\begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = M\mathbf{u}$. ◀

▶ Remark 4.2. In the proofs $\delta_\top(x, \overline{M}^B)$, $\delta^1_\wedge(x, y.(t_1\ y))$, and $\delta^2_\wedge(x, z.(t_2\ z))$, the variable $x$ occurs in one argument of the symbols $\delta_\top$, $\delta^1_\wedge$, and $\delta^2_\wedge$, but not in the other. In contrast, in the proof $\delta^1_\wedge(x, y.(t_1\ y)) \boldsymbol{+} \delta^2_\wedge(x, z.(t_2\ z))$, it occurs in both arguments of the symbol $\boldsymbol{+}$. Thus, these proofs are well-typed in the system of Figure 2.

▶ Remark 4.3. The rules

$$\begin{aligned} \delta_\top(a.\star, t) &\longrightarrow a \bullet t & \delta^1_\wedge(\langle t, u\rangle, x.v) &\longrightarrow (t/x)v \\ (\lambda x.t)\ u &\longrightarrow (u/x)t & \delta^2_\wedge(\langle t, u\rangle, x.v) &\longrightarrow (u/x)v \end{aligned}$$

were introduced as cut reduction rules.

Now, these rules appear to be also matrix calculation rules.

▶ **Example 4.4** (Matrices with two colums and two lines). The matrix $\begin{pmatrix} a & c \\ b & d \end{pmatrix}$ is expressed as the proof

$$t = \lambda x.(\delta^1_\wedge(x, y.\delta_\top(y, \langle a.\star, b.\star\rangle)) \boldsymbol{+} \delta^2_\wedge(x, z.\delta_\top(z, \langle c.\star, d.\star\rangle)))$$

And applying the rules of Figure 3, we get

$$t\ \langle e.\star, f.\star\rangle \longrightarrow^* \langle (a \times e + c \times f).\star, (b \times e + d \times f).\star\rangle$$

## 5 Linearity

We now prove the converse: if $A, B \in \mathcal{V}$, then each proof $t$ of $A \Rightarrow B$ expresses a linear function, that is

$$t\ (u \boldsymbol{+} v) \equiv (t\ u) \boldsymbol{+} (t\ v) \qquad \text{and} \qquad t\ (a \bullet u) \equiv a \bullet (t\ u)$$

A first idea could be to generalize this statement and prove that these properties hold for all closed proofs $t$, whatever their type. But this generalization is too strong. For example, if $A = \top$ and $B = (\top \Rightarrow \top) \Rightarrow \top$, $t = \lambda x.\lambda y.(y\ x)$ is a proof of $A \Rightarrow B$, but

$$t\ (1.\star \boldsymbol{+} 2.\star) \longrightarrow^* \lambda y.(y\ 3.\star) \qquad \text{and} \qquad t\ 1.\star \boldsymbol{+} t\ 2.\star \longrightarrow^* \lambda y.((y\ 1.\star) \boldsymbol{+} (y\ 2.\star))$$

and these two irreducible proofs are different. So we will prove that these properties hold when $A$ is arbitrary and $B \in \mathcal{V}$.

▶ Remark 5.1. The fact that we want all proofs of $\top \Rightarrow \top$ to be linear functions from $\mathcal{S}$ to $\mathcal{S}$ explains why the symbol $\top$ must be multiplicative. If it were additive, the proposition $\top \Rightarrow \top$ would have the proof $f = \lambda x.\ (1.\star)$ that is not linear as $f\ (1.\star \boldsymbol{+} 1.\star) \longrightarrow^* 1.\star \not\equiv 2.\star\ {}^*\!\!\longleftarrow (f\ 1.\star) \boldsymbol{+} (f\ 1.\star)$.

▶ Remark 5.2. The fact that we want all proofs of $\top \Rightarrow \top$ to be linear functions from $\mathcal{S}$ to $\mathcal{S}$ explains why the rule sum must be additive. If it were multiplicative, the proposition $\top \Rightarrow \top$ would have the proof $g = \lambda x.\ (x \boldsymbol{+} 1.\star)$ that is not linear as $g\ (1.\star \boldsymbol{+} 1.\star) \longrightarrow^* 3.\star \not\equiv 4.\star\ {}^*\!\!\longleftarrow (g\ 1.\star) \boldsymbol{+} (g\ 1.\star)$.

### 5.1 Size of a proof

The proof of the linearity theorem proceeds by induction on the size of the proof, and the first part of this proof is the definition of such a size function $\mu$. Our goal could be to build a size function such that if $t$ is proof of $B$ in a context $\Gamma, x : A$ and $u$ is a proof of $A$, then $\mu((u/x)t) = \mu(t) + \mu(u)$. This would be the case, for the usual notion of size, if $x$ had exactly one occurrence in $t$. But, due to additive connectives, the variable $x$ may have zero, one, or several occurrences in $t$.

First, as the rule $\bot$-e is additive, it may happen that $\delta_\bot(t)$ is a proof in the context $\Gamma, x : A$, and $x$ has no occurrence in $t$. Thus, we lower our expectations to $\mu((u/x)t) \leq \mu(t) + \mu(u)$, which is sufficient for the linearity theorem.

Then, as the rules $\boldsymbol{+}$, $\wedge$-i, and $\vee$-e rules are additive, if $u \boldsymbol{+} v$ is proof of $B$ in a context $\Gamma, x : A$, $x$ may occur both in $u$ and in $v$. And the same holds for the proofs $\langle u, v\rangle$, and $\delta_\vee(t, x.u, y.v)$. In these cases, we modify the definition of the size function and take $\mu(t \boldsymbol{+} u) = 1 + \max(\mu(t), \mu(u))$, instead of $\mu(t \boldsymbol{+} u) = 1 + \mu(t) + \mu(u)$, etc. making the function $\mu$ a mix between a size function and a depth function. Note that the depth function itself cannot be used, as Lemma 5.8 does not hold for the depth function.

This leads to the following definition.

▶ **Definition 5.3** (Size of a proof).

- $\mu(x) = 0$
- $\mu(t \boldsymbol{+} u) = 1 + \max(\mu(t), \mu(u))$
- $\mu(a \bullet t) = 1 + \mu(t)$

- $\mu(a.\star) = 1$
- $\mu(\delta_\top(t, u)) = 1 + \mu(t) + \mu(u)$,
- $\mu(\delta_\bot(t)) = 1 + \mu(t)$
- $\mu(\lambda x.t) = 1 + \mu(t)$
- $\mu(t\ u) = 1 + \mu(t) + \mu(u)$
- $\mu(\langle t, u\rangle) = 1 + \max(\mu(t), \mu(u))$
- $\mu(\delta^1_\wedge(t, y.u)) = 1 + \mu(t) + \mu(u)$
- $\mu(\delta^2_\wedge(t, y.u)) = 1 + \mu(t) + \mu(u)$
- $\mu(\mathit{inl}(t)) = 1 + \mu(t)$
- $\mu(\mathit{inr}(t)) = 1 + \mu(t)$
- $\mu(\delta_\vee(t, y.u, z.v)) = 1 + \mu(t) + \max(\mu(u), \mu(v))$

▶ **Lemma 5.4.** *If $\Gamma, x : A \vdash t : B$ and $\Delta \vdash u : B$ then $\mu((u/x)t) \leq \mu(t) + \mu(u)$.*

▶ **Example 5.5.** Let $t = \delta_\bot(y)$ and $u = 1.\star$. We have $y : \bot, x : \top \vdash t : C$, $\mu(t) = 1$, $\mu(u) = 1$ and $\mu((u/x)t) = 1$. Thus $\mu((u/x)t) \leq \mu(t) + \mu(u)$.

As a corollary, we get a similar size preservation theorem for reduction.

▶ **Lemma 5.6.** *If $t \longrightarrow u$, then $\mu(t) \geq \mu(u)$.*

## 5.2 Elimination contexts

The second part of the proof is a standard generalization of the notion of head variable. In the $\lambda$-calculus, we can decompose a term $t$ as a sequence of applications $t = u\ v_1\ \dots\ v_n$, with terms $v_1, \dots, v_n$ and a term $u$, which is not an application. Then $u$ may either be a variable, in which case it is the head variable of the term, or an abstraction.

In a similar way, any proof in the $\mathcal{L}^{\mathcal{S}}$-calculus can be decomposed into a sequence of elimination rules, forming an elimination context, and a proof $u$ that is either a variable, an introduction, a sum, or a product.

▶ **Definition 5.7** (Elimination context)**.** *An elimination context is a proof with a single free variable, written _, that is in the language*

$$K = _ \mid \delta_\top(K, u) \mid \delta_\bot(K) \mid K\ t \mid \delta^1_\wedge(K, x.r) \mid \delta^2_\wedge(K, x.r) \mid \delta_\vee(K, x.r, y.s)$$

*where $u$ is a closed proof, $FV(r) \subseteq \{x\}$, and $FV(s) \subseteq \{y\}$.*

In the case of elimination contexts, Lemma 5.4 can be strengthened.

▶ **Lemma 5.8.** $\mu(K\{t\}) = \mu(K) + \mu(t)$

Note that in Example 5.5, $(_/x)t$ is not a context as _ does not occur in it.

▶ **Lemma 5.9** (Decomposition of a proof)**.** *If $t$ is an irreducible proof such that $x : C \vdash t : A$, then there exist an elimination context $K$, a proof $u$, and a proposition $B$ such that $_ : B \vdash K : A$, $x : C \vdash u : B$, $u$ is either the variable $x$, an introduction, a sum, or a product, and $t = K\{u\}$.*

A final lemma shows that, in the same way we can always decompose a non-empty list into a smaller list and its last element, we can always decompose an elimination context $K$ different from _ into an elimination context $K_1$ and a last elimination rule $K_2$.

▶ **Lemma 5.10** (Decomposition of an elimination context). *If $K$ is a elimination context such that $_ : A \vdash K : B$ and $K \neq _$, then $K$ has the form $K_1\{K_2\}$, and*

- *if $A = \top$, then $K_2$ has the form $\delta_\top(_, t)$,*
- *if $A = \bot$, then $K_2$ has the form $\delta_\bot(_)$,*
- *if $A = B \Rightarrow C$, then $K_2$ has the form $_\ t$,*
- *if $A = B \wedge C$, then $K_2$ has the form $\delta^1_\wedge(_, x.t)$ or $\delta^2_\wedge(_, x.t)$,*
- *if $A = B \vee C$, then $K_2$ has the form $\delta_\vee(_, x_1.t_1, x_2.t_2)$.*

## 5.3 Linearity

We now have the tools to prove the linearity theorem. Instead of proving the theorem for a closed proof $t$ of $A \Rightarrow B$, it is more convenient to prove it for a proof $t$ of $B$ in the context $x : A$. The result for the proofs of $A \Rightarrow B$ is Corollary 5.12.

▶ **Theorem 5.11** (Linearity). *For every proposition $A$, proposition $B \in \mathcal{V}$, proofs $t$, $u_1$, and $u_2$, such that $x : A \vdash t : B$, $t$ is irreducible, $\vdash u_1 : A$, and $\vdash u_2 : A$. Then*

$$t\{u_1 + u_2\} \equiv t\{u_1\} + t\{u_2\} \qquad \text{and} \qquad t\{a \bullet u_1\} \equiv a \bullet t\{u_1\}$$

**Proof (Sketch).** The proof proceeds by induction on the size $\mu(t)$ of the proof $t$, but the organization of the cases is complex. We first consider the different forms for $t$: it can be a variable, a sum, a product, an introduction, or an elimination. If it is an introduction, as $B \in \mathcal{V}$, it must be a pair. The key point here is that taking $B \in \mathcal{V}$, we avoid the case of the abstraction that would lead to a failure, as show by the counter-example above.

The case where $t$ is an elimination leads to a second case analysis. We first use Lemma 5.9 to decompose the proof $t$ into an elimination context $K$ and a proof $v$. Then we consider the different possible forms of $v$: it can be neither an introduction nor an elimination, hence it is a variable, a sum, or a product.

Finally, in the case it is a variable, we have a third case analysis: we use Lemma 5.10 to decompose $K$ into a context $K'$ and a last elimination rule and we consider the different possible cases for this last elimination rule.

In all these cases, we use various cases of the Lemma 3.4 to prove the convertibility of the proofs, and the Lemmas 5.4, 5.6, and 5.8 to show that the induction hypothesis applies to smaller proofs. ◀

▶ **Corollary 5.12.** *Let $A$ be a proposition and $B \in \mathcal{V}$. Let $t$ be a closed proof of $A \Rightarrow B$ and $u$ and $v$ be closed proofs of $A$. Then*

$$t\ (u + v) \equiv (t\ u) + (t\ v) \qquad \text{and} \qquad t\ (a \bullet u) \equiv a \bullet (t\ u)$$

▶ Remark 5.13. As we have seen, Corollary 5.12 does not generalize when $B \notin \mathcal{V}$. For example, $t = \lambda x.\lambda y.(y\ x)$ is a closed irreducible form of $\top \Rightarrow (\top \Rightarrow \top) \Rightarrow \top$, but the proofs $t\ (1.\star + 2.\star)$ and $t\ 1.\star + t\ 2.\star$ are not convertible. Indeed

$$t\ (1.\star + 2.\star) \longrightarrow^* \lambda y.(y\ 3.\star) \qquad \text{and} \qquad t\ 1.\star + t\ 2.\star \longrightarrow^* \lambda y.((y\ 1.\star) + (y\ 2.\star))$$

and the two irreducible proofs $\lambda y.(y\ 3.\star)$ and $\lambda y.((y\ 1.\star) + (y\ 2.\star))$ are different.

Yet, these two proofs are observationally equivalent: if $B \in \mathcal{V}$ and $s$ is a closed proof of $((\top \Rightarrow \top) \Rightarrow \top) \Rightarrow B$

$$s\ (t\ (1.\star + 2.\star)) \equiv s\ (t\ 1.\star + t\ 2.\star)$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash [t,u] : A \odot B} \odot\text{-i}$$

$$\frac{\Gamma \vdash t : A \odot B \quad \Delta, x : A \vdash u : C}{\Gamma, \Delta \vdash \delta^1_\odot(t, x.u) : C} \odot\text{-e1}$$

$$\frac{\Gamma \vdash t : A \odot B \quad \Delta, x : B \vdash u : C}{\Gamma, \Delta \vdash \delta^2_\odot(t, x.u) : C} \odot\text{-e2}$$

$$\frac{\Gamma \vdash t : A \odot B \quad \Delta, x : A \vdash u : C \quad \Delta, y : B \vdash v : C}{\Gamma, \Delta \vdash \delta_\odot(t, x.u, y.v) : C} \odot\text{-e}$$

**Figure 4** The deduction rules of the $\mathcal{L}\odot^{\mathcal{S}}$-calculus.

Indeed, applying Corollary 5.12 with the proof $\lambda x\ (s\ (t\ x))$, we obtain that the first proof is convertible with $(s\ (t\ 1.\star)) + (s\ (t\ 2.\star))$ and applying it to $s$ we obtain that the second is convertible with this same proof.

▶ **Corollary 5.14.** *Let $A, B \in \mathcal{V}$, such that $d(A) = m$ and $d(B) = n$, and $t$ be a closed proof of $A \Rightarrow B$. Then the function $F$ from $\mathcal{S}^m$ to $\mathcal{S}^n$, defined as $F(\mathbf{u}) = \underline{t\ \overline{\mathbf{u}}^A}$ is linear.*

## 5.4 No-cloning

In the PL$^{\mathcal{S}}$-calculus, that is in the proof language of propositional logic extended with interstitial rules and scalars, the cloning function from $\mathcal{S}^2$ to $\mathcal{S}^4$, mapping $\left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right)$ to $\left(\begin{smallmatrix} a^2 \\ ab \\ ab \\ b^2 \end{smallmatrix}\right)$ can be expressed with the proof of $(\top \wedge \top) \Rightarrow ((\top \wedge \top) \wedge (\top \wedge \top))$

$$\begin{aligned}&\lambda x.\delta^1_\wedge(x, y.\delta^1_\wedge(x, y_1.\langle\langle\delta_\top(y, y_1), 0.\star\rangle, \langle 0.\star, 0.\star\rangle\rangle) + \delta^2_\wedge(x, z_1.\langle\langle 0.\star, \delta_\top(y, z_1)\rangle, \langle 0.\star, 0.\star\rangle\rangle))\\ &\quad +\\ &\quad \delta^2_\wedge(x, z.\delta^1_\wedge(x, y_2.\langle\langle 0.\star, 0.\star\rangle, \langle\delta_\top(z, y_2), 0.\star\rangle\rangle) + \delta^2_\wedge(x, z_2.\langle\langle 0.\star, 0.\star\rangle, \langle 0.\star, \delta_\top(z, z_2)\rangle\rangle)))\end{aligned}$$

This proof cannot be expressed in $\mathcal{L}^{\mathcal{S}}$-calculus, as it uses twice an elimination symbol of the conjunction with the variable $x$ occurring in both arguments.

Moreover, by Corollary 5.14, this function cannot be expressed as a proof of the proposition $(\top \wedge \top) \Rightarrow ((\top \wedge \top) \wedge (\top \wedge \top))$ in the $\mathcal{L}^{\mathcal{S}}$-calculus, because it is not linear.

# 6 The $\mathcal{L}\odot^{\mathcal{S}}$-calculus and its application to quantum computing

## 6.1 The $\mathcal{L}\odot^{\mathcal{S}}$-calculus

The $\mathcal{L}\odot^{\mathcal{S}}$-calculus is obtained by adding the symbols $[,]$, $\delta^1_\odot$, $\delta^2_\odot$, $\delta_\odot$, the deduction rules of Figure 4, and the reduction rules of Figure 5, to the $\mathcal{L}^{\mathcal{S}}$-calculus. It is similar to the $\odot^{\mathcal{S}}$-calculus [5, long version] except that its typing rules are linear.

## 6.2 Quantum computing

Like the $\odot^{\mathbb{C}}$-calculus, the $\mathcal{L}\odot^{\mathbb{C}}$-calculus, with a reduction strategy restricting the reduction of $\delta_\odot([t,u], x.v, y.w)$ to the cases where $t$ and $u$ are closed irreducible proofs, can be used to express quantum algorithms. The following reproduces the Section 4 of [5, long version], focusing on the differences due to linearity.

$$
\begin{aligned}
\delta^1_{\odot}([t,u],x.v) &\longrightarrow (t/x)v\\
\delta^2_{\odot}([t,u],x.v) &\longrightarrow (u/x)v\\
\delta_{\odot}([t,u],x.v,y.w) &\longrightarrow (t/x)v\\
\delta_{\odot}([t,u],x.v,y.w) &\longrightarrow (u/y)w\\
\\
[t,u] \mathbin{\pmb{+}} [v,w] &\longrightarrow [t \mathbin{\pmb{+}} v, u \mathbin{\pmb{+}} w]\\
a \bullet [t,u] &\longrightarrow [a \bullet t, a \bullet u]
\end{aligned}
$$

**Figure 5** The reduction rules of the $\mathcal{L}\odot^{\mathcal{S}}$-calculus.

Bits can be expressed as proofs of the proposition $\top \vee \top$: $\mathbf{0} = inl(1.\star)$ and $\mathbf{1} = inr(1.\star)$. The test operation was defined in [5, long version] as

$$if(t,u,v) = \delta_{\vee}(t, x.u, y.v)$$

where $x$ and $y$ are variables not occurring in $u$ and $v$. But this proof is not linear, so we define it as

$$if(t,u,v) = \delta_{\vee}(t, x.\delta_{\top}(x,u), y.\delta_{\top}(y,v))$$

Note that $if(\mathbf{0},u,v) \longrightarrow 1 \bullet u$ and $if(\mathbf{1},u,v) \longrightarrow 1 \bullet v$.

Then, we express the vectors, like in Section 3, except that we use the connective $\odot$ instead of $\wedge$. For instance, the vector $\left(\begin{smallmatrix} a \\ b \end{smallmatrix}\right)$ is not expressed as the proof $\langle a.\star, b.\star\rangle$ but as the proof $[a.\star, b.\star]$, etc. In particular $n$-qubit, for $n \geq 1$, are expressed, in the basis $|0\dots00\rangle, |0\dots01\rangle, \dots |1\dots11\rangle$, as elements of $\mathbb{C}^{2^n}$, that is as proofs of the proposition $\mathcal{Q}_n$ defined by induction on $n$ as follows: $\mathcal{Q}_0 = \top$ and $\mathcal{Q}_{n+1} = \mathcal{Q}_n \odot \mathcal{Q}_n$.

If $t$ is a closed irreducible proof of $\mathcal{Q}_n$, we define the square of the norm $\|t\|^2$ of $t$ by induction on $n$.

- If $n = 0$, then $t = a.\star$ and we take $\|t\|^2 = |a|^2$.
- If $n = n' + 1$, then $t = [u_1, u_2]$ and we take $\|t\|^2 = \|u_1\|^2 + \|u_2\|^2$.

We take the convention that any closed irreducible proof $u$ of $\mathcal{Q}_n$, expressing a non-zero vector $\underline{u} \in \mathbb{C}^{2^n}$, is an alternative expression of the $n$-qubit $\frac{u}{\|\underline{u}\|}$. For example, the qubit $\frac{1}{\sqrt{2}}.|0\rangle + \frac{1}{\sqrt{2}}.|1\rangle$ is expressed as the proof $[\frac{1}{\sqrt{2}}.\star, \frac{1}{\sqrt{2}}.\star]$, but also as the proof $[1.\star, 1.\star]$.

Matrices are expressed as in Section 4.

Like in [5, long version], thanks to the reduction strategy, probabilities can be assigned to the non-deterministic reductions of closed proofs of the form $\delta_{\odot}(u, x.v, y.w)$, that is proofs of the form $\delta_{\odot}([u_1,u_2], x.v, y.w)$.

If $u_1$ and $u_2$ are closed irreducible proofs of of $\mathcal{Q}_n$ and $\|u_1\|^2$ and $\|u_2\|^2$ are not both 0, then we assign the probability $\frac{\|u_1\|^2}{\|u_1\|^2+\|u_2\|^2}$ to the reduction

$$\delta_{\odot}([u_1,u_2], x.v, y.w) \longrightarrow (u_1/x)v$$

and the probability $\frac{\|u_2\|^2}{\|u_1\|^2+\|u_2\|^2}$ to the reduction

$$\delta_{\odot}([u_1,u_2], x.v, y.w) \longrightarrow (u_2/y)w$$

If $\|u_1\|^2 = \|u_2\|^2 = 0$, or $u_1$ and $u_2$ are proofs of propositions of a different form, we assign any probability, for example $\frac{1}{2}$, to both reductions.

If $n$ is a non-zero natural number, we can define the measurement operator $\pi_n$, measuring the first qubit of an $n$-qubit, as the proof

$$\pi_n = \lambda x.\delta_\odot(x, y.[y, 0_{\mathcal{Q}_{n-1}}], z.[0_{\mathcal{Q}_{n-1}}, z])$$

of the proposition $\mathcal{Q}_n \Rightarrow \mathcal{Q}_n$.

Indeed, if $t$ is a closed irreducible proof of $\mathcal{Q}_n$ of the form $[u_1, u_2]$, such that $\|t\|^2 = \|u_1\|^2 + \|u_2\|^2 \neq 0$, expressing the state of an $n$-qubit, then the proof $\pi_n\ t$ of the proposition $\mathcal{Q}_n$ reduces, with probabilities $\frac{\|u_1\|^2}{\|u_1\|^2+\|u_2\|^2}$ and $\frac{\|u_2\|^2}{\|u_1\|^2+\|u_2\|^2}$ to $[u_1, 0_{\mathcal{Q}_{n-1}}]$ and to $[0_{\mathcal{Q}_{n-1}}, u_2]$, that are the states of the $n$-qubit, after the partial measure of the first qubit.

Note that, as the $\mathcal{L}\odot^{\mathcal{S}}$-calculus is purely linear, it cannot express the measurement operator $\lambda x.\delta_\odot(x, y.\mathbf{0}, z.\mathbf{1})$ that returns the "classical" result of the measure and that could be expressed in the $\odot^{\mathcal{S}}$-calculus. Typing this measurement operator would require to extend the type system to express that, in the premises $\Delta, A \vdash C$ and $\Delta, B \vdash C$ of the rule $\odot$-e, the hypotheses $A$ and $B$ may be weakened.

Instead, our measurement operators return the state of the full system after the measure. In the first case, this state is a linear combination of the first $2^{n-1}$ vectors $|00\dots0\rangle \dots |01\dots1\rangle$ of the basis: those starting with a 0, in the second, this state is a linear combination of the last $2^{n-1}$ vectors $|10\dots0\rangle \dots |11\dots1\rangle$ of the basis: those starting with a 1. In the first case, the result of the measurement is $|0\rangle$, in the second it is $|1\rangle$.

As we have a representation of linear functions and measurement operators, we can express in the $\mathcal{L}\odot^{\mathcal{S}}$-calculus, all quantum algorithms, for instance Deutsch's algorithm.

### 6.3 Linearity

The main motivation for introducing this linear variant of the $\odot^{\mathbb{C}}$-calculus was to prove a linearity theorem for this calculus. But, the $\mathcal{L}\odot^{\mathbb{C}}$-calculus contains the $\delta_\odot$ symbol, that enables to express measurement operators, which are not linear.

Thus, our linearity theorem should be that using the $\delta_\odot$ symbol is the only way to construct a non-linear function. In other words, that, in the fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-calculus excluding the $\delta_\odot$ symbol, only linear functions can be expressed. But, if $\odot$-e rule is excluded, $\odot$ is just another conjunction, and this fragment of the $\mathcal{L}\odot^{\mathcal{S}}$-logic is the $\mathcal{L}^{\mathcal{S}}$-logic with two copies of the conjunction. As a corollary of the Corollary 5.14, only linear functions can be expressed in this calculus and cloning cannot.

## 7 Conclusion

We can now attempt a possible answer to the question stated in the introduction: in which way must propositional logic be extended or restricted, so that its proof language becomes a quantum programming language. This answer is in four parts: we need to extend it with interstitial rules, scalars, and the connective $\odot$, and we need to restrict it by making it linear.

We obtain this way the $\mathcal{L}\odot^{\mathcal{S}}$-logic that addresses both the question of linearity and, for instance, avoids cloning, and that of the information-erasure, non-reversibility, and non-determinism of the measurement.

Another issue is to restrict the logic further so that linear functions are unitary. We can either enforce unitarity, following the methods of [1, 6, 7], or let these unitarity constraints as properties of the program that must be proved for each program, rather than enforced by the type system.

We may also wish to make this quantum representation more compositional, by considering some form of tensor product. Indeed, for example in Lineal [2] the tensor product is just the standard encoding of pairs, since in Lineal pairs are, by construction, bilinear, so a pair of superpositions (which are constructed with the symbol + in that language) such as $\langle\alpha_1.|0\rangle + \alpha_2.|1\rangle, \beta_1.|0\rangle + \beta_2.|1\rangle\rangle$ would reduce to $\alpha_1\beta_1.\langle|0\rangle,|0\rangle\rangle + \alpha_1\beta_2.\langle|0\rangle,|1\rangle\rangle + \alpha_2\beta_1.\langle|1\rangle,|0\rangle\rangle + \alpha_2\beta_2.\langle|1\rangle,|1\rangle\rangle$ which represents the four-dimensional vector obtained by the tensor product. It is not the case in the $\mathcal{L}\odot^{\mathcal{S}}$-calculus, since the pair does not commute with the sup, so $\langle[\alpha_1.\star,\alpha_2.\star],[\beta_1.\star,\beta_2.\star]\rangle$ is in normal form. However, we could encode a tensor product $\otimes_{n,m}$ as a proof term of $Q_n \wedge Q_m \Rightarrow Q_{n\times m}$, whose size depends on $n$ and $m$, or, to be more in line with Lineal, where there is no dependency on the size, just introduce a new proof term for a new rule

$$\frac{\Gamma \vdash t : \top^n \quad \Delta \vdash r : \top^m}{\Gamma\Delta \vdash t \otimes r : \top^{n\times m}}\ \text{tens}$$

with the following reduction rules:

$$[t,r]\otimes s \longrightarrow [t\otimes s, r\otimes s] \qquad \text{and} \qquad \alpha.\star \otimes t \longrightarrow \alpha \bullet t$$

This way, $[\alpha_1.\star,\alpha_2.\star]\otimes[\beta_1.\star,\beta_2.\star] \longrightarrow^* [[\alpha_1\beta_1.\star,\alpha_1\beta_2.\star],[\alpha_2\beta_1.\star,\alpha_2\beta_2.\star]]$.

Also, it may be interesting to study if a tensor connective could be added, with some notion of equivalence where $Q_n \otimes Q_m \equiv Q_{n\times m}$, relating the $\odot$ connective with the $\otimes$ connective. We left this study for future work.

## References


1. T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings of LICS 2005*, pages 249–258. IEEE, 2005.
2. P. Arrighi and G. Dowek. Lineal: A linear-algebraic lambda-calculus. *Logical Methods in Computer Science*, 13(1), 2017.
3. R. Blute. Hopf algebras and linear logic. *Mathematical Structures in Computer Science*, 6(2):189–217, 1996.
4. B. Coecke and A. Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning.* Cambridge University Press, 2017. `doi:10.1017/9781316219317`.
5. A. Díaz-Caro and G. Dowek. A new connective in natural deduction, and its application to quantum computing. In A. Cerone and P. Csaba Ölveczky, editors, *Prooceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 12819 of *Lecture Notes in Computer Science*, pages 175–193. Springer, 2021. Long version accessible at `arXiv:2012.08994`.
6. A. Díaz-Caro, M. Guillermo, A. Miquel, and B. Valiron. Realizability in the unitary sphere. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019)*, pages 1–13, 2019.
7. A. Díaz-Caro and O. Malherbe. Quantum control in the unitary sphere: Lambda-$S_1$ and its categorical model. Draft at `arXiv:2012.05887`, 2020.
8. A. Díaz-Caro, G. Dowek, and J.P. Rinaldi. Two linearities for quantum computing in the lambda calculus. *Biosystems*, 2019.
9. Th. Ehrhard. On Köthe sequence spaces and linear logic. *Mathematical Structures in Computer Science*, 12(5):579–623, 2002.
10. J.-Y. Girard. Linear logic. *Theoreoretical Computer Science*, 50:1–102, 1987.
11. J.-Y. Girard. Coherent banach spaces: A continuous denotational semantics. *Theoretical Computer Science*, 227(1-2):275–297, 1999.

**12** P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

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

**14** M. Zorzi. On quantum lambda calculi: a foundational perspective. *Mathematical Structures in Computer Science*, 26(7):1107–1195, 2016.

# A Graphical Proof Theory of Logical Time

**Matteo Acclavio**
Department of Computer Science, University of Luxembourg, Luxembourg

**Ross Horne**
Department of Computer Science, University of Luxembourg, Luxembourg

**Sjouke Mauw**
Department of Computer Science, University of Luxembourg, Luxembourg

**Lutz Straßburger**
Inria-Saclay, Palaiseau, France

## Abstract

Logical time is a partial order over events in distributed systems, constraining which events precede others. Special interest has been given to series-parallel orders since they correspond to formulas constructed via the two operations for "series" and "parallel" composition. For this reason, series-parallel orders have received attention from proof theory, leading to pomset logic, the logic BV, and their extensions. However, logical time does not always form a series-parallel order; indeed, ubiquitous structures in distributed systems are beyond current proof theoretic methods. In this paper, we explore how this restriction can be lifted. We design new logics that work directly on graphs instead of formulas, we develop their proof theory, and we show that our logics are conservative extensions of the logic BV.

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

**Keywords and phrases** proof theory, causality, deep inference

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.22

## 1 Introduction

This paper is about the design of graphical proof systems that have an expressive power beyond logics constrained by the syntax of formulas. We explain, in this introduction, our original observations that compelled us to explore this untapped region in logic, whilst exploring a proof theory of logical time. Logical time is a partial order constraining events. It is employed because some events have to precede others (for instance due to read/write dependencies) and because synchronising clocks is infeasible in distributed systems.

A special case is given by series-parallel orders [39, 18, 6] which gave rise to a family of non-commutative logics, including pomset logic [36] and BV [19]. A series-parallel order is a partial order that can be constructed by composing smaller components, in series ($\triangleleft$) and in parallel ($\parr$). For example, the following two series-parallel orders $a \longrightarrow b \longrightarrow c \longrightarrow d$ and $a, b \to c, d$ (with $a \to c$, $a \to d$, $b \to c$, $b \to d$) correspond to the formulas $(a \triangleleft b) \triangleleft (c \triangleleft d)$ and $(a \parr b) \triangleleft (c \parr d)$, respectively.

Pomset logic and BV give series-parallel orders a first-class status inside a logical system. Both logics extend the core of linear logic, hence feature a multiplicative conjunction ($\otimes$) that is de Morgan dual to $\parr$. Implication, $A \multimap B$ is internalised as $A^\perp \parr B$, where $A^\perp$ is linear negation, with respect to which sequential composition is self dual, that is, $(A \triangleleft B)^\perp = A^\perp \triangleleft B^\perp$. For example, in both BV and pomset logic we have $(a \triangleleft b) \triangleleft (c \triangleleft d) \multimap (a \parr b) \triangleleft (c \parr d)$, and hence the above-mentioned series-parallel orders are related by implication.

© Matteo Acclavio, Ross Horne, Sjouke Mauw, and Lutz Straßburger;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 22; pp. 22:1–22:25
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Modelling logical time, where the causal relation "happens before" constrains events in a distributed system in such a way that they certainly precede others [29], has always been a core motivation for BV [8]. Implication can be a sound tool for reasoning about behavioural preorders between processes such as simulation [11, 26, 24, 25], and it can be used to devise powerful notions of multiparty compatibility and subtyping in session types [23].

However, all the above-mentioned work is limited by the series-parallel restriction, that is, on orders which can be decomposed using series and parallel compositions. Yet, not all patterns of causality can be represented by series-parallel orders. Consider for example read-write dependencies, where shapes such as

$$\begin{array}{ccc} \mathtt{read}_x & \longrightarrow & \mathtt{write}_{h(x)} \\ & \searrow & \\ \mathtt{read}_y & \longrightarrow & \mathtt{write}_{g(x,y)} \end{array} \qquad (1)$$

can occur. In this diagram there are two data write operations where one is dependent on two previous read operations, while the other is dependent on a single read. The diagram in (1) above is not a series-parallel order and indeed is out-of-scope of BV and pomset logic. In contrast, the diagram is in scope of general process models, such as Pratt's original pomsets and event structures [33, 34].

Clearly, such non-series-parallel structures arise ubiquitously. As a running example, let us consider consumer-producer queues which are used to preserve message order when composing asynchronous distributed components, as depicted in Figure 1. In these diagrams, the labels on the nodes represent the events of enqueueing or dequeueing four particular messages ($a$, $b$, $c$, and $d$). Then, an enqueue event must happen before its corresponding dequeue event. Secondly, if one enqueue event happens before another enqueue event, then the corresponding dequeue events occur in the same order. This preserves a first-in-first-out order for messages exchanged using the queue. Finally, queues with capacity $n$ have the restriction that the message $i$ must be dequeued before message $i + n$ is enqueued.

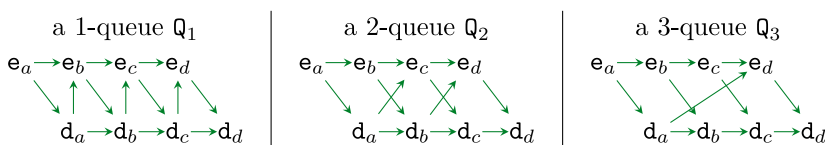



**Figure 1** Causality patterns for consumer-producer $n$-queues, where $n$ is the bound on the number of messages that can be enqueued. Nodes labelled by $\mathtt{e}_x$ and $\mathtt{d}_x$ respectively represent the enqueuing and dequeuing of the message $x$. In all three queue types we only represent the first four messages ($a$, $b$, $c$, and $d$) inserted into the queue.

The edges in the diagrams in Figure 1 should not be seen as the possible paths when "executing" the system, but they describe the dependencies that have to be met before executing each event. If there are no edges between a given pair of events and if all dependencies of these two events are met, then they can be executed in any order. In other words, they describe the Hasse diagram of the induced partial order. In this way, one can see that dependencies are relaxed from left to right in Figure 1, permitting more concurrency.

The first example is a 1-queue, allowing at most one message to be enqueued at any time. The result is a simple total order over events, alternating between enqueue and dequeue events, represented by the formula $\mathtt{e}_a \lhd \mathtt{d}_a \lhd \mathtt{e}_b \lhd \mathtt{d}_b \lhd \mathtt{e}_c \lhd \mathtt{d}_c \lhd \mathtt{e}_d \lhd \mathtt{d}_d$. This helps explain why some verification frameworks for distributed systems, e.g. [15, 12], assume 1-queues as a

simplified approximation of queues. The 2-queue, allowing two messages to be buffered, is also a series-parallel order, represented by $\mathtt{e}_a \lhd (\mathtt{e}_b \parr \mathtt{d}_a) \lhd (\mathtt{e}_c \parr \mathtt{d}_b) \lhd (\mathtt{e}_d \parr \mathtt{d}_c) \lhd \mathtt{d}_d$. Hence we can use BV or pomset logic to reason about these queues, for example proving that the 1-queue implies the 2-queue, serving as a proof that a 2-queue can simulate behaviours of a 1-queue.[1] In the third example, up to three messages can be buffered in the queue. However, this 3-queue is not a series-parallel order,[2] nor is any other $n$-queue with $n > 2$. Therefore the existing proof-theoretical tools cannot be used to reason about them.

This is the main motivation for this paper. Can we design proof systems that work directly on graphs that are not bound by the series-parallel restriction, i.e., go beyond formulas?

This problem turned out to be more challenging than expected. A first step was communicated in [1], which devised a minimal logic, called GS, over general undirected graphs, where the symmetric edges generalise the multiplicative connectives of linear logic, without the presence of directed edges representing logical time. The proof theory of GS requires deep inference, where inference rules may be applied deep in any context, not just at the root connective of a formula as for example in the sequent calculus. For developing a structural proof theory in the absence of formulas, we employed the notion of *modular decomposition* from graph theory, which enables any graph to be represented as a tree of *prime graphs*.

The main contribution of this paper is a graphical logic that generalizes both, GS and BV. Following the recent discovery that BV and pomset logic are not the same [32], we also made the observation that there is not one canonical choice about what is the "right" logic. We present here two proof systems GV and $\mathsf{GV}^{\mathsf{sl}}$, that both (i) are analytic (i.e., suitable for proof search) and (ii) obey cut elimination. In this respect, we can indeed speak of a proper proof theory. We present our logics using *open deduction*, which is a deep inference formalism. Furthermore, we show (iii) that both logics, GV and $\mathsf{GV}^{\mathsf{sl}}$, are conservative extensions of GS and of BV, and (iv) that the provability problem in GV and in $\mathsf{GV}^{\mathsf{sl}}$ is **NP**-complete.

**Structure of the paper.** We begin by recalling some preliminary notions about graphs and their modular decomposition in Section 2 and some preliminaries about deep inference and open deduction in Section 3. In Section 4 we introduce the inference rules for our proof systems and show some of their properties. Then, in Section 5 and Section 6 we prove cut elimination and some of its consequences, including the results on conservativity and complexity mentioned above.

## 2 Preliminaries on Graphs and their Modular Decomposition

A ***directed graph*** $G = \langle V_G, \overset{G}{\curvearrowright} \rangle$ is a set $V_G$ of *vertices* equipped with an irreflexive binary *edge relation* $\overset{G}{\curvearrowright} \subseteq V_G \times V_G$. An ***undirected graph*** $G = \langle V_G, \overset{G}{\frown} \rangle$ is a set $V_G$ of *vertices* equipped with an irreflexive and symmetric binary *edge relation* $\overset{G}{\frown} \subseteq V_G \times V_G$. A ***mixed graph*** (or simply ***graph***) is a triple $G = \langle V_G, \overset{G}{\frown}, \overset{G}{\curvearrowright} \rangle$ where $\langle V_G, \overset{G}{\frown} \rangle$ is an undirected graph and $\langle V_G, \overset{G}{\curvearrowright} \rangle$ is a directed graph, such that $\overset{G}{\frown} \cap \overset{G}{\curvearrowright} = \varnothing$. In a mixed graph we define the following additional edge-relations:

[1] The proof for the implication $\mathtt{Q}_1 \multimap \mathtt{Q}_2$ is shown in Figure 3.

[2] It suffices to remark that the vertices in $\{\mathtt{e}_b, \mathtt{e}_c, \mathtt{e}_d, \mathtt{d}_a\}$ induce an N-shaped subgraph similar to the one from Equation (1).

$$\overset{G}{\not\curvearrowright} = \{(v,w) \mid v \neq w \text{ and } (v,w) \notin \overset{G}{\curvearrowright}\} \qquad \overset{G}{\not\frown} = \{(v,w) \mid v \neq w \text{ and } (v,w) \notin \overset{G}{\frown}\}$$
$$\overset{G}{\curvearrowleft} = \{(w,v) \mid (v,w) \in \overset{G}{\curvearrowright}\} \qquad \overset{G}{\smile} = \{(v,w) \mid v \overset{G}{\not\frown} w \text{ and } v \overset{G}{\not\curvearrowright} w \text{ and } w \overset{G}{\not\curvearrowright} v\} \tag{2}$$

We say that a graph $G$ is ***n*-colour** (for $n \leq 3$) if at most $n$ of the three edge relations $\smile$ (white edges), $\curvearrowright$ (green edges) and $\frown$ (red edges) are non-empty. In particular, we say that $G$ is a ***complete graph*** if $\overset{G}{\smile} = \varnothing$, a directed graph if $\overset{G}{\frown} = \varnothing$, and an undirected graph if $\overset{G}{\curvearrowright} = \varnothing$. The empty graph is denoted as $\varnothing$.

In the following we may omit the index/superscript $G$ when it is clear from the context. When drawing a graph we draw $v$—$w$ whenever $v \frown w$, we draw $v \rightarrow w$ whenever $v \curvearrowright w$, and we draw no edge at all whenever $v \smile w$.

▶ **Definition 1.** *Let $G$ and $H$ be graphs. If $V' \subseteq V_G$, then we define the **subgraph induced by $V'$** as the graph $G|_{V'} = \langle V', \overset{G}{\frown} \cap (V' \times V'), \overset{G}{\curvearrowright} \cap (V' \times V') \rangle$. We write $H \sqsubseteq G$ if there is an injective homomorphism $f$ from $H$ to $G$ such that $f(H)$ is an induced subgraph of $G$, and we say that $G$ is **$H$-free** whenever $H \sqsubseteq G$ does not hold.*

As shown previously [1, 9], graphs can be used as operators to compose graphs, playing a similar role to connectives in formulas.

▶ **Definition 2.** *Let $G$ be a graph with $n$ vertices $V_G = \{v_1, \ldots, v_n\}$ and let $H_1, \ldots, H_n$ be $n$ graphs, with disjoint vertices. We define the **composition of $H_1, \ldots, H_n$ via $G$** as the graph $G(\!|H_1, \ldots, H_n|\!)$ obtained by replacing each vertex $v_i$ of $G$ by the graph $H_i$. That is, the graph with vertices the union of the vertices of $H_1, \ldots, H_n$ and with an edge $x \frown y$ (resp. $x \curvearrowright y$) if either $x$ and $y$ are in the same $H_i$ and $x \overset{H_i}{\frown} y$ (resp. $x \overset{H_i}{\curvearrowright} y$), or $x \in V_{H_i}$ and $y \in V_{H_j}$ for $i \neq j$ and $v_i \overset{G}{\frown} v_j$ (resp. $v_i \overset{G}{\curvearrowright} v_j$). Formally,*

$$V_{G(\!|H_1,\ldots,H_n|\!)} = \bigcup_{1 \leq i \leq n} V_{H_i}$$
$$\overset{G(\!|H_1,\ldots,H_n|\!)}{\frown} = \left( \bigcup_{1 \leq i \leq n} \overset{H_i}{\frown} \right) \cup \left\{ (x,y) \mid x \in V_{H_i} \text{ and } y \in V_{H_j} \text{ and } v_i \overset{G}{\frown} v_j \text{ and } i,j \in \{1, \ldots, n\} \right\}$$
$$\overset{G(\!|H_1,\ldots,H_n|\!)}{\curvearrowright} = \left( \bigcup_{1 \leq i \leq n} \overset{H_i}{\curvearrowright} \right) \cup \left\{ (x,y) \mid x \in V_{H_i} \text{ and } y \in V_{H_j} \text{ and } v_i \overset{G}{\curvearrowright} v_j \text{ and } i,j \in \{1, \ldots, n\} \right\}$$

In developing our proof systems, we use the notion of *modular decomposition* [16, 27, 22, 30, 31, 13, 7] in order to assign to each graph a tree-like structure where leaves are single vertex graphs, and modules are sub-trees.[3]

▶ **Definition 3.** *A **module** of a graph $G$ is a subset $M$ of $V_G$ such that $x R z$ iff $y R z$ for any $x, y \in M$, $z \in V_G \setminus M$ and $R \in \{\frown, \curvearrowright, \curvearrowleft\}$. We say that a module $M$ is **trivial** if $M = \varnothing$, $M = V_G$, or $M = \{x\}$ for some $x \in V_G$. A graph $G$ is **prime** if $|V_G| > 1$ and all its modules are trivial.*

▶ **Notation 4.** *In this paper we identify a module $M$ of a graph $G$ and its induced subgraph $G|_M$. We introduce the following notation for relevant prime graphs on 2, 3 and 4 vertices, respectively:*

[3] The notion of module we use here coincides with the one of *clans* from the literature on 2-structures [13].

$\otimes$ : •—•  $\triangleleft$ : •→•  $\parr$ : •  •

$\mathsf{T}_{RG}$ : •—•→•  $\mathsf{T}_{GR}$ : •→•—•

$\mathsf{T}^{W}_{GG}$ : •→•→•  $\mathsf{T}^{R}_{GG}$ : •→•→•  $\mathsf{T}^{G}_{GG}$ : •→•→• (3)

$\mathsf{P}_4$ : •—•—•—•  $\mathsf{N}$ : •→•←•→•

*We use the following notation for the composition via (prime) graphs with two vertices:*

$$G \parr H = \parr(\!|G,H|\!) \qquad G \otimes H = \otimes(\!|G,H|\!) \qquad G \triangleleft H = \triangleleft(\!|G,H|\!) \quad .$$

Prime graphs play a special role in modular decomposition and can be considered as generalized (non-decomposable) logic connectives in the sense of [17, 10, 3].

▶ **Theorem 5** ([16, 27, 13])**.** *Let $G$ be a graph with at least two vertices. Then there are some non-empty graphs $H_1, \ldots, H_n$ and a prime graph $P$ such that $G = P(\!|H_1, \ldots, H_n|\!)$.*

▶ **Corollary 6.** *Each graph $G$ admits a **modular decomposition** by means of prime graphs and single-vertex graphs. Such a decomposition is unique modulo associativity of $\otimes$, $\parr$ and $\triangleleft$, and prime graphs isomorphisms.*

▶ **Example 7.** Consider the graph on the left below.

$c$, $e$, $d$, $f$, $a \rightarrow b$, $g \rightarrow h \leftarrow i \rightarrow j$   $c$—$d$, $e$—$f$, $a \rightarrow b$, $g \rightarrow h \leftarrow i \rightarrow j$ (4)

The representation on the right above relies on modular decomposition to reduce the number of edges to draw: the vertices inside a rectangle belong to a same module. Therefore an edge touching a rectangle represents a bundle of edges (of the same type) connecting with each vertex in the module. We can associate a ***modular decomposition tree*** to each graph in the same way we associate an abstract syntax tree to a formula. Below is a tree representation of the modular decomposition of the graph above and its formula-like denotation, which uses the prime graphs from (3) and the composition notation from Definition 2.

$\mathsf{P}_4$; $\triangleleft$ ($a$, $b$), $\otimes$ ($c$, $d$), $\otimes$ ($e$, $f$), $\mathsf{N}$ ($g$, $h$, $i$, $j$)

$$\mathsf{P}_4(\!|(a \triangleleft b), (c \otimes d), (e \otimes f), \mathsf{N}(\!|g,h,i,j|\!)|\!) \qquad (5)$$

Indeed, prime graphs can be thought as primitive operators or connectives for graphical logic (see Section 9 of [2]), and the notion of module can be seen as a generalization of subformulas.

In this paper, we use the graphical representation on the right of (4) and the formula-like representation on the right of (5) interchangeably.

▶ **Definition 8.** *A prime graph $P$ is a **connector** of a graph $G$ (or **$P$-connector** if we want to specify the prime graph $P$) if there is an occurrence of the graph $P$ in the modular decomposition of $G$.*

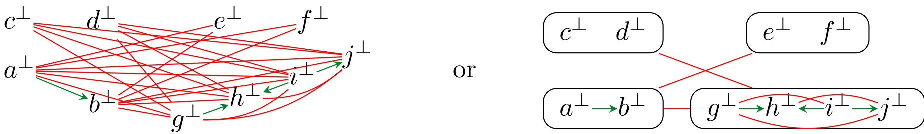
For example, in the graph in Example 7 above, we have two connectors $\otimes$, one $\triangleleft$, one $\mathsf{N}$ and one $\mathsf{P}_4$.

We assume graphs are labelled. That is, each vertex $v$ of a graph $G$ carries a label $\ell_G(v)$ selected from a label set $\mathcal{L}$. In particular in this paper we assume $\mathcal{L} = \mathcal{A} \cup \mathcal{A}^\perp$ where $\mathcal{A}$ is a set of atoms $\{a, b, c, \ldots\}$ and $\mathcal{A}^\perp$ is the set of negated atoms $\{a^\perp, b^\perp, c^\perp, \ldots\}$.

▶ **Definition 9.** *Let $G = \langle V_G, \overset{G}{\frown}, \overset{G}{\curvearrowright} \rangle$ be a graph, we define its* ***dual*** *graph $G^\perp = \langle V_{G^\perp}, \overset{G^\perp}{\frown}, \overset{G^\perp}{\curvearrowright} \rangle$ with $V_{G^\perp} = V_G$ and $\overset{G^\perp}{\frown} = \overset{G}{\smile}$ and $\overset{G^\perp}{\curvearrowright} = \overset{G}{\curvearrowright}$. The label of each vertex in $V_{G^\perp}$, is the dual of the label of the corresponding vertex in $G$, that is, if $\ell_G(v) = x$, then $\ell_{G^\perp}(v) = x^\perp$. For this purpose we assume negation to be involutive, that is, $x^{\perp\perp} = x$. The* ***implication*** *$G \multimap H$ is defined as $G^\perp \parr H$.*

The dual graph operation above flips labels on vertices, exchanges $\frown$ (red) and $\smile$ (white) symmetric edges, but preserves $\curvearrowright$ (green) directed edges.

▶ **Example 10.** The graph $\mathsf{P_4}^\perp(\!|(a^\perp \triangleleft b^\perp), (c^\perp \parr d^\perp), (e^\perp \parr f^\perp), \mathsf{N}^\perp(\!|g^\perp, h^\perp, i^\perp, j^\perp|\!)|\!)$ is the dual of the graph from Example 7 and is represented as follows:



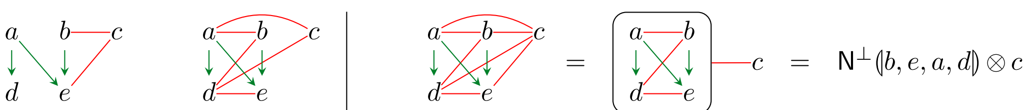

In this work we are mainly interested in how vertices are labelled, and not in the identity of the underlying vertex. For this reason we rely on the notion of graph isomorphism.

▶ **Definition 11.** *Let $G$ and $G'$ be two graphs. We say that $G$ and $G'$ are* ***isomorphic*** *if there is a bijection $f \colon V_G \to V_{G'}$ such that $x \frown y$ iff $f(x) \frown f(y)$ and $x \curvearrowright y$ iff $f(x) \curvearrowright f(y)$ for any $x, y \in V_G$. If the graphs are labelled, then we additionally require that $\ell(v) = \ell(f(v))$. If $G$ and $G'$ are isomorphic we write $G \sim_f G'$, or simply $G \sim G'$ if the isomorphism $f$ is clear form the context.*

▶ **Definition 12.** *A* ***web*** *is a graph such that $\curvearrowright$ is transitive and such that each connector occurring in its modular decomposition is a 2-colour prime graph. A* ***relation web*** *is a web such that each connector is in $\{\parr, \triangleleft, \otimes\}$. A* ***cograph*** *is a relation web with $\curvearrowright = \varnothing$. A* ***series-parallel order*** *is a relation web with $\frown = \varnothing$.*

It is worth noticing that the above graphs may be characterised by ruling out the presence of specific induced subgraphs (see, e.g., [37, 19, 14]). In particular, cographs are $\mathsf{P_4}$-free undirected graphs, series-parallel orders are $\mathsf{N}$-free directed graphs with $\curvearrowright$ being transitive, and relation webs are $\mathsf{T}_{GR}$-free, $\mathsf{P_4}$-free, and $\mathsf{N}$-free graphs with transitive $\curvearrowright$. Note that requiring $\curvearrowright$ to be transitive is equivalent to requiring that a graph is $\mathsf{T}^W_{GG}$-free, $\mathsf{T}^G_{GG}$-free and $\mathsf{T}^R_{GG}$-free. Moreover, graphs with no 3-colour connectors are by definition $\mathsf{T}_{GR}$-free and $\mathsf{T}_{RG}$-free, however the converse does not hold.

▶ **Example 13.** The two graphs below on the left are not webs since they are three-colour prime graphs, while the complete graph on the right is a web since its modular decomposition only contains 2-colour connectors.




▶ **Notation 14.** *In drawing webs we may omit directed edges whenever they are derivable from the drawn ones via the transitivity of $\curvearrowright$. This means, we can represent the web $u \triangleleft v \triangleleft w$ as $u \to v \to w$, in which we implicitly assume the directed edge from $u$ to $w$. For example, the web $\mathsf{Q}_1$ in Figure 1 can be represented as a single thread $\mathsf{e}_a \to \mathsf{d}_a \to \mathsf{e}_b \to \mathsf{d}_b \to \mathsf{e}_c \to \mathsf{d}_c \to \mathsf{e}_d \to \mathsf{d}_d$ .*

▶ Remark 15. The operation of composing graphs via a graph (Definition 2) is closed with respect to the sets of cographs, series-parallel orders, relation webs and webs. That is, if all $H_1, \ldots, H_n$ and $G$ are cographs (resp. series-parallel orders, relation webs, webs), then so is $G(\!|H_1, \ldots, H_n|\!)$.

The correspondence between cographs and formulas containing only connectives for conjunction and disjunction is well-known since the 60s [16, 27]. We here recall the correspondence between relation webs and the formulas for the non-commutative logic BV from [19, 35].

▶ **Definition 16.** *The set of* BV*-formulas is defined by a countable set of propositional* ***atoms*** $\mathcal{A} = \{a, b, \ldots\}$*, a special symbol* $\circ$ *called* ***unit****, and the following grammar.*

$$\phi, \psi := a \mid a^\perp \mid \circ \mid \phi \parr \psi \mid \phi \lhd \psi \mid \phi \otimes \psi$$

*The binary connectives* $\parr$*,* $\lhd$ *and* $\otimes$ *are called* ***par****,* ***seq*** *and* ***tensor****. Par and tensor correspond to disjunction and conjunction. We associate to each formula* $\phi$ *a relation web* $[\![\phi]\!]$ *defined as follows:*

$$\begin{array}{ccc} [\![\circ]\!] = \varnothing & [\![a]\!] = a & [\![a^\perp]\!] = a^\perp \\ [\![\phi \lhd \psi]\!] = [\![\phi]\!] \lhd [\![\psi]\!] & [\![\phi \parr \psi]\!] = [\![\phi]\!] \parr [\![\psi]\!] & [\![\phi \otimes \psi]\!] = [\![\phi]\!] \otimes [\![\psi]\!] \end{array} \tag{6}$$

*We identify an atom* $a$ *(resp. its negation* $a^\perp$*) with a single vertex graph labelled by the same atom* $a$ *(resp.* $a^\perp$*).*

▶ **Theorem 17** ([35, 19])**.** *A graph* $G$ *is a relation web iff there is a* BV*-formula* $\phi$ *such that* $G = [\![\phi]\!]$*.*

We will make use of this result later in Section 4, when formulating the proof system for BV as a proof system on relation webs rather than a proof system on formulas.

## 3 Preliminaries on Open Deduction

Open deduction [20] is a proof formalism based on deep inference [5]. It has originally been defined for formulas, but it is abstract enough such that it can equally well be used for graphs, as already done in [2]. We begin here by defining the notion of *context*, which is a graph with a hole into which another graph can be plugged, generalising the similar notion of context for formulas.

▶ **Definition 18** (Context)**.** *A* ***context*** $\mathcal{C}[\square]$ *is a graph containing a single occurrence of a special vertex* $\square$*. If* $\mathcal{C}[\square]$ *is a context and* $G$ *a graph, we define* $\mathcal{C}[G]$ *as the graph obtained by replacing* $\square$ *by* $G$*, that is, the graph obtained by replacing* $\square$ *by (the modular decomposition of)* $G$ *in the modular decomposition of* $\mathcal{C}[\square]$*.*

Note that the notion of context is strongly connected to the one of module: $M$ is a module of a graph $G$ iff $G = \mathcal{C}[M]$ for some context $\mathcal{C}[\square]$.

▶ **Definition 19.** *An* ***inference system*** S *is a set of inference rules (as for example shown in Figure 2). A* ***derivation*** $\mathcal{D}$ *in* S *with premise* $G$ *and conclusion* $H$ *is denoted* $\begin{array}{c} G \\ \mathcal{D} \| \mathsf{S} \\ H \end{array}$ *and is defined inductively as follows:*

- *Every graph* $G$ *is a (****trivial****) derivation (also denoted* $G$*) with premise* $G$ *and conclusion* $G$*.*

- *Every instance of a rule* $\mathsf{r}\,\frac{G}{H}$ *in* $\mathsf{S}$ *is a derivation with premise* $G$ *and conclusion* $H$*.*
- *If* $\mathcal{D}_1$ *is a derivation with premise* $G_1$ *and conclusion* $H_1$*, and* $\mathcal{D}_2$ *is a derivation with premise* $G_2$ *and conclusion* $H_2$*, and* $H_1 \sim_f G_2$*, then the composition of* $\mathcal{D}_1$ *and* $\mathcal{D}_2$ *is a derivation* $\mathcal{D}_2 \circ_f \mathcal{D}_1$ *denoted as below on the left.*

$$\begin{array}{c} G_1 \\ \mathcal{D}_1 \| \mathsf{S} \\ H_1 \\ f \cdots\cdots \\ G_2 \\ \mathcal{D}_2 \| \mathsf{S} \\ H_2 \end{array} \quad \Bigg| \quad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \mathsf{S} \\ H_1 \\ \cdots\cdots \\ G_2 \\ \mathcal{D}_2 \| \mathsf{S} \\ H_2 \end{array} \quad \text{or} \quad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \mathsf{S} \\ H_1 \\ \mathcal{D}_2 \| \mathsf{S} \\ H_2 \end{array} \quad \text{or} \quad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \mathsf{S} \\ G_2 \\ \mathcal{D}_2 \| \mathsf{S} \\ H_2 \end{array} \tag{7}$$

*If* $f$ *is clear from context we may simply write* $\mathcal{D}_2 \circ \mathcal{D}_1$ *and denote it as above on the right in order to ease readability. However, even if the isomorphism* $f$ *is not written, we always assume it is part of the derivation and explicitly given.*
- *If* $G$ *is a graph with* $n$ *vertices and* $\mathcal{D}_1, \ldots, \mathcal{D}_n$ *are derivations with premise* $G_i$ *and conclusion* $H_i$ *for each* $i \in \{1, \ldots, n\}$*, then* $G(\!|\mathcal{D}_1, \ldots, \mathcal{D}_n|\!)$ *is a derivation with premise* $G(\!|G_1, \ldots, G_n|\!)$ *and conclusion* $G(\!|H_1, \ldots, H_n|\!)$ *denoted as below on the left (if* $G = \otimes$ *or* $G = \lhd$ *or* $G = \parr$ *we may write the derivations as below on the right).*

$$G\left(\!\!\left|\begin{array}{c} G_1 \\ \mathcal{D}_1 \| \mathsf{S} \\ H_1 \end{array}, \ldots, \begin{array}{c} G_n \\ \mathcal{D}_n \| \mathsf{S} \\ H_n \end{array}\right|\!\!\right) \quad \Bigg| \quad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array} \otimes \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array} \qquad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array} \lhd \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array} \qquad \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array} \parr \begin{array}{c} G_1 \\ \mathcal{D}_1 \| \\ H_1 \end{array}$$

*A* ***proof*** *in* $\mathsf{S}$ *is a derivation in* $\mathsf{S}$ *whose premise is* $\varnothing$*. A graph* $G$ *is* ***provable*** *in* $\mathsf{S}$ *iff there is a proof in* $\mathsf{S}$ *with conclusion* $G$*. We denote this as* $\vdash_{\mathsf{S}} G$*.*

Thus, if $\begin{array}{c} G \\ \mathcal{D} \| \mathsf{S} \\ H \end{array}$ is a derivation and $\mathcal{C}[\Box]$ a context, then we have a derivation $\mathcal{C}\left[\begin{array}{c} G \\ \mathcal{D} \| \mathsf{S} \\ H \end{array}\right] = \begin{array}{c} \mathcal{C}[G] \\ \mathcal{C}[\mathcal{D}] \| \mathsf{S} \\ \mathcal{C}[H] \end{array}$.

## 4 Proof Systems on Webs

In this section, we recall the proof systems BV from [19] (that we here formulate as a proof system on relation webs) and GS from [2] (which is a proof system on undirected graphs), and we introduce two new proof system on webs using the rules from Figure 2.[4]

$$\begin{array}{lcl} \mathsf{BV} & = & \{\mathsf{ai}{\downarrow}, \mathsf{s}, \mathsf{q}_{\mathsf{BV}}{\downarrow}\} \\ \mathsf{GS} & = & \{\mathsf{ai}{\downarrow}, \mathsf{s}_{\parr}, \mathsf{p}{\downarrow}\} \end{array} \qquad \begin{array}{lcl} \mathsf{GV} & = & \{\mathsf{ai}{\downarrow}, \mathsf{s}_{\parr}, \mathsf{s}_{\otimes}, \mathsf{p}{\downarrow}, \mathsf{q}{\downarrow}, \mathsf{qm}\} \\ \mathsf{GV}^{\mathsf{sl}} & = & \{\mathsf{ai}{\downarrow}, \mathsf{s}_{\parr}, \mathsf{s}_{\otimes}, \mathsf{p}{\downarrow}, \mathsf{q}{\downarrow}, \mathsf{qm}, \mathsf{sl}\} \end{array} \tag{8}$$

[4] Note that in [2, 1] the system GS is defined with the rule $\mathsf{s.sw}_{\parr}\,\frac{\mathcal{C}[M]}{\mathcal{C}[\varnothing] \parr M}\,{\scriptstyle M \neq \varnothing}$ instead of $\mathsf{s}_{\parr}$. However it is easy to show that $\mathsf{s.sw}_{\parr}$ is derivable using $\mathsf{s}_{\parr}$.

$$\mathsf{ai}\!\downarrow \frac{\varnothing}{a^\perp \parr a} \qquad \mathsf{ai}\!\uparrow \frac{a^\perp \otimes a}{\varnothing}$$

$$\mathsf{q_{BV}}\!\downarrow \frac{(N_1 \parr N_3) \lhd (N_2 \parr N_4)}{(N_1 \lhd N_2) \parr (N_3 \lhd N_4)} \qquad \mathsf{s} \frac{(M_1 \parr N) \otimes M_2}{M_1 \parr (N \otimes M_2)} \qquad \mathsf{q_{BV}}\!\uparrow \frac{(N_1 \otimes N_2) \otimes (N_3 \otimes N_4)}{(N_1 \otimes N_3) \lhd (N_2 \otimes N_4)}$$

$$\mathsf{s}_{\parr} \frac{P(\!|M_1, \ldots, M_{i-1}, M_i \parr N, M_{i+1}, \ldots M_n|\!)}{M_i \parr P(\!|M_1, \ldots, M_{i-1}, N, M_{i+1}, \ \ldots, M_n|\!)} \qquad \mathsf{s}_{\otimes} \frac{M_i \otimes P(\!|M_1, \ldots, M_{i-1}, N, M_{i+1}, \ldots, M_n|\!)}{P(\!|M_1, \ldots, M_{i-1}, M_i \otimes N, M_{i+1}, \ldots, M_n|\!)}$$

$$\mathsf{p}\!\downarrow \frac{(M_1 \parr N_1) \otimes \cdots \otimes (M_n \parr N_n)}{R^\perp(\!|M_1, \ldots, M_n|\!) \parr R(\!|N_1, \ldots, N_n|\!)} \qquad \mathsf{p}\!\uparrow \frac{R(\!|M_1, \ldots, M_n|\!) \otimes R^\perp(\!|N_1, \ldots, N_n|\!)}{(M_1 \otimes N_1) \parr \cdots \parr (M_n \otimes N_n)}$$

$$\mathsf{q}\!\downarrow \frac{Q^\perp(\!|L_1 \parr J_1, \ldots, L_n \parr J_n|\!)}{Q^\perp(\!|L_1, \ldots, L_n|\!) \parr Q(\!|J_1, \ldots, J_n|\!)} \qquad \mathsf{q}\!\uparrow \frac{Q(\!|L_1, \ldots, L_n|\!) \otimes Q^\perp(\!|J_1, \ldots, J_n|\!)}{Q(\!|L_1 \otimes J_1, \ldots, L_n \otimes J_n|\!)}$$

$$\mathsf{qm} \frac{Q(\!|L_1 \parr J_1, \ldots, L_n \parr J_n|\!)}{Q(\!|L_1, \ldots, L_n|\!) \parr Q(\!|J_1, \ldots, J_n|\!)}$$

$$\mathsf{sl} \frac{Q(\!|M_1, \ldots, M_k, \varnothing, \ldots, \varnothing|\!) \lhd Q(\!|\varnothing, \ldots, \varnothing, M_{k+1}, \ldots, M_n|\!)}{Q(\!|M_1, \ldots, M_n|\!)}$$

*For all rules we assume $P, Q$ and $R$ prime webs with $\overset{R}{\curvearrowright} = \varnothing$, $\overset{Q}{\frown} = \varnothing$, and $M_i \neq \varnothing \neq L_i \parr J_i$ for all $i$.*
*For the rule* sl *we also require $y \not\curvearrowright x$ for all $x \in M_i, y \in M_j$ with $0 \leq i \leq k < j \leq n$.*

**Figure 2** Inference rules for our family of proof systems on webs.

Following the deep inference tradition, the rules come in pairs, a down-rule (marked with ↓) and an up-rule (marked with ↑), which are dual to each other. The proof systems in (8) above only contain the down-rules, as the up-rules are admissible, which we will show in the next section.[5]

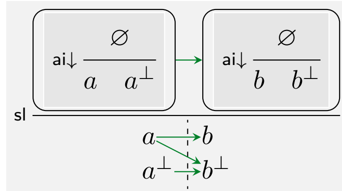
The rules ai are called ***atomic interactions*** and allow to derive the atomic implication $a \multimap a$. The rules s (called ***switch***), $\mathsf{q_{BV}}\!\downarrow$ and $\mathsf{q_{BV}}\!\uparrow$ are the rules obtained by adapting the standard proof system BV on formulas to relation webs. When generalising s from relation webs to webs, this rule admits two formulations $\mathsf{s}_{\parr}$ and $\mathsf{s}_{\otimes}$, which are respectively called ***switch par*** and ***switch tensor***. Intuitively, $\mathsf{s}_{\parr}$ allows to move a connected component $M_i$ inside a context of the shape $\mathcal{C}[\square] = P(\!|M_1, \ldots, M_{i-1}, N \parr \square, M_{i+1}, \ldots M_n|\!)$. Dually, $\mathsf{s}_{\otimes}$ allows to extract a module $M_i$ from a context $\mathcal{C}[\square] = P(\!|M_1, \ldots, M_{i-1}, N \otimes \square, M_{i+1}, \ldots M_n|\!)$.[6] The rules p are called ***prime graph*** rules.[7] Each instance of $\mathsf{p}\!\downarrow$ creates disjunctions of the modules in a prime connector $R$ and the corresponding modules in an occurrence of its dual

[5] The rules sl and qm are in fact down-rules, but we omitted the ↓ as we do not need the corresponding up-rules in this paper.
[6] From inspecting the rules, one can observe that $\mathsf{s}_{\parr}$ and $\mathsf{s}_{\otimes}$ are dual to each other. And indeed in simpler systems like BV and GS, the $\mathsf{s}_{\otimes}$ is admissible like all other "up-rules". However, for GV and $\mathsf{GV}^{\mathsf{sl}}$, this is no longer the case (see Example 44 in the appendix).
[7] Differently from [1, 2] in this paper we drop the side condition $|V_P| \geq 4$ in p-rules. This choice restrains the set of rules required to simulate the general i-rules (see Lemma 24). Note that this makes the s-rule also a special case of $\mathsf{p}\!\downarrow$.

$$
\mathsf{5{\times}q{\downarrow}}\ \frac{
\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{e}_a^\perp \parr \mathsf{e}_a}
\ \lhd\ 
\mathsf{s}_\parr \frac{\dfrac{\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{e}_b^\perp \parr \mathsf{e}_b}}{\left(\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{d}_a^\perp \parr \mathsf{d}_a} \lhd \mathsf{e}_b^\perp\right) \parr \mathsf{e}_b}}{(\mathsf{d}_a^\perp \lhd \mathsf{e}_b^\perp) \parr \mathsf{d}_a \parr \mathsf{e}_b}
\ \lhd\ 
\mathsf{s}_\parr \frac{\dfrac{\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{e}_c^\perp \parr \mathsf{e}_c}}{\left(\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{d}_b^\perp \parr \mathsf{d}_b} \lhd \mathsf{e}_c^\perp\right) \parr \mathsf{e}_c}}{(\mathsf{d}_b^\perp \lhd \mathsf{e}_c^\perp) \parr \mathsf{d}_b \parr \mathsf{e}_c}
\ \lhd\ 
\mathsf{s}_\parr \frac{\dfrac{\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{e}_d^\perp \parr \mathsf{e}_d}}{\left(\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{d}_c^\perp \parr \mathsf{d}_c} \lhd \mathsf{e}_d^\perp\right) \parr \mathsf{e}_d}}{(\mathsf{d}_c^\perp \lhd \mathsf{e}_d^\perp) \parr \mathsf{d}_c \parr \mathsf{e}_d}
\ \lhd\ 
\mathsf{ai{\downarrow}}\frac{\varnothing}{\mathsf{d}_d^\perp \parr \mathsf{d}_d}
}{(\mathsf{e}_a^\perp \lhd \mathsf{d}_a^\perp \lhd \mathsf{e}_b^\perp \lhd \mathsf{d}_b^\perp \lhd \mathsf{e}_c^\perp \lhd \mathsf{d}_c^\perp \lhd \mathsf{e}_d^\perp \lhd \mathsf{d}_d^\perp) \parr (\mathsf{e}_a \lhd (\mathsf{d}_a \parr \mathsf{e}_b) \lhd (\mathsf{d}_b \parr \mathsf{e}_c) \lhd (\mathsf{d}_c \parr \mathsf{e}_d) \lhd \mathsf{d}_d)}
$$

**Figure 3** Proof of $\mathsf{Q}_1 \multimap \mathsf{Q}_2$ from the introduction. This is a correct proof in BV, in GV, and in $\mathsf{GV}^{\mathsf{sl}}$.

$R^\perp$ and puts them in conjunction. The rules q and qm are called ***q-rules*** and generalise the two rules $\mathsf{q}_{\mathsf{BV}}{\downarrow}$ and $\mathsf{q}_{\mathsf{BV}}{\uparrow}$ from BV to general prime webs with an empty set of ⌒-edges. The rule sl is called ***slice*** and is yet another generalisation the $\mathsf{q}_{\mathsf{BV}}{\downarrow}$-rule. It formalises the idea that during proof search we aim at slicing a directed graph into a "before" and an "after" part by introducing additional ↷-edges. This mimics a longstanding idea in distributed systems: a partial order is a representation of many possible linear orders in which concurrent events may occur [29]. For example, below is a proof of a small web provable in $\mathsf{GV}^{\mathsf{sl}}$ but not provable in GV.



Other examples of derivations are given in Figure 3, Figure 4, Figure 5, and Figure 6.

▶ Remark 20. The rules p↓, p↑, q↓ and q↑ are crucial to obtain a proof system containing only atomic interactions (see Proposition 28). These rules gather two dual connectors by merging their set of edges, therefore obtaining a complete graph. In particular, the formulation of the rule q↓ exploits the fact that if $Q$ is a prime web such that $\overset{Q}{\curvearrowright} \neq \varnothing$, then either $Q$ or $Q^\perp$ is a complete graph. This will be crucial for proving the results in this paper. However, as Example 13 shows, this fact is not true for general 3-colour prime graphs. The question whether the results on webs, that are presented in this paper, can be generalized to arbitrary graphs is an open problem.

Let us now show that GV and $\mathsf{GV}^{\mathsf{sl}}$ do indeed extend BV beyond the restriction of relation webs (i.e., formulas).

▶ **Lemma 21.** *Let $A$ and $B$ be relation webs.*

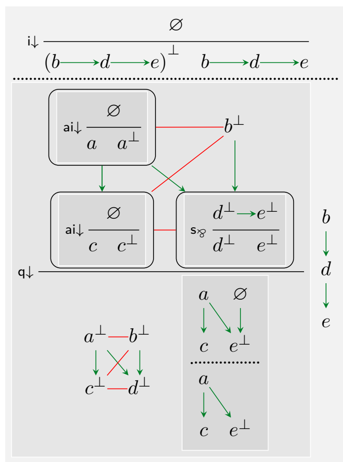
$$
\text{if } \mathsf{s}\frac{B}{A}\text{, then } \mathsf{s}_\parr\frac{B}{A} \qquad \text{if } \mathsf{q}_{\mathsf{BV}}{\downarrow}\frac{B}{A}\text{, then } \mathsf{r}\frac{B}{A} \qquad \text{if } \mathsf{q}_{\mathsf{BV}}{\uparrow}\frac{B}{A}\text{, then } \mathsf{t}\frac{B}{A} \tag{9}
$$

*where* $\mathsf{r} \in \{\mathsf{q}{\downarrow}, \mathsf{s}_\parr\}$ *and* $\mathsf{t} \in \{\mathsf{q}{\uparrow}, \mathsf{s}_\otimes\}$.

**Proof.** This can be seen immediately by inspecting the rules in Figure 2. ◀

We define the additional (non-atomic) ***interaction*** rules

$$
\mathsf{i}{\downarrow}\frac{\varnothing}{G^\perp \parr G} \quad \text{and} \quad \mathsf{i}{\uparrow}\frac{G^\perp \otimes G}{\varnothing} \tag{10}
$$



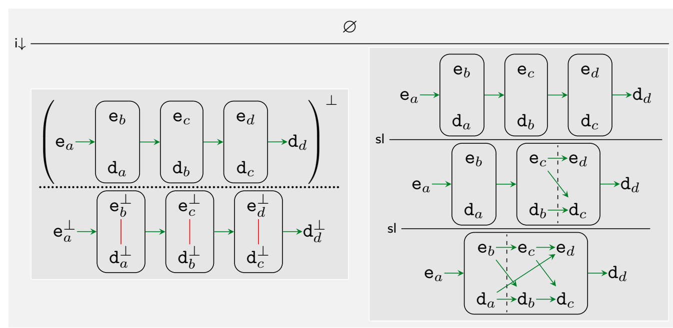




**Figure 4** Examples of proofs in GV and $\mathsf{GV}^{\mathsf{sl}}$. As the conclusions are not relation webs, these proofs cannot be carried out in BV. The left proof is valid in GV and $\mathsf{GV}^{\mathsf{sl}}$. The right proof uses the slice rule and is only valid in $\mathsf{GV}^{\mathsf{sl}}$. It proves $\mathtt{Q}_2 \multimap \mathtt{Q}_3$ from the introduction.

The $\mathsf{i}\uparrow$ rule is the deep inference equivalent to the cut rule from sequent calculus and proving its admissibility is equivalent to proving the cut-elimination result. We recall some admissibility results for rules in BV and GS from the literature where we notice that the admissibility of cuts can be achieved by proving the admissibility of up-rules.

▶ **Theorem 22** ([19]). *Let $G$ be a relation web. Then $\vdash_{\mathsf{BV}\cup\{\mathsf{i}\uparrow\}} G$ iff $\vdash_{\mathsf{BV}\cup\{\mathsf{ai}\uparrow,\mathsf{q}_{\mathsf{BV}}\uparrow\}} G$ iff $\vdash_{\mathsf{BV}} G$.*

▶ **Theorem 23** ([1]). *Let $G$ be an undirected graph. Then $\vdash_{\mathsf{GS}\cup\{\mathsf{i}\uparrow\}} G$ iff $\vdash_{\mathsf{GS}\cup\{\mathsf{ai}\uparrow,\mathsf{s}_{\otimes},\mathsf{p}\uparrow\}} G$ iff $\vdash_{\mathsf{GS}} G$.*

In the next section we are going to generalize these two theorems to GV and $\mathsf{GV}^{\mathsf{sl}}$. For doing so, we end this section with the following two auxiliary lemmas.

▶ **Lemma 24.** *Let $M_1,\dots,M_n,N_1,\dots,N_n$ and $G$ be webs such that $|V_G| = n$. Then there is a complete graph $\mathbb{C}$ with $|V_{\mathbb{C}}| = n$ such that there are derivations*

$$\begin{array}{c}\mathbb{C}(\!|M_1 \parr N_1,\dots,M_n \parr N_n|\!)\\ \mathcal{D}^{\downarrow}\,\Big\Vert\,\{\mathsf{s}_{\otimes},\mathsf{p}\downarrow,\mathsf{q}\downarrow\}\\ G^{\perp}(\!|M_1,\dots,M_n|\!) \parr G(\!|N_1,\dots,N_n|\!)\end{array} \quad \text{and} \quad \begin{array}{c}G^{\perp}(\!|M_1,\dots,M_n|\!) \otimes G(\!|N_1,\dots,N_n|\!)\\ \mathcal{D}^{\uparrow}\,\Big\Vert\,\{\mathsf{s}_{\parr},\mathsf{p}\uparrow,\mathsf{q}\uparrow\}\\ \mathbb{C}^{\perp}(\!|M_1 \otimes N_1,\dots,M_n \otimes N_n|\!)\end{array}$$

*Moreover, if $M_1,\dots,M_n,N_1,\dots,N_n$ are non-empty, then $\mathcal{D}^{\downarrow}$ contains no $\mathsf{s}_{\otimes}$ and $\mathcal{D}^{\uparrow}$ contains no $\mathsf{s}_{\parr}$.*

**Proof.** We proceed by induction on the modular decomposition of $G$. If $G$ is atomic, then we conclude by letting $\mathbb{C}$ be atomic and $\mathcal{D}^{\downarrow}$ trivial. Otherwise, without loss of generality we can assume $M_i \parr N_i \neq \varnothing$ for all $i \in \{1,\dots,n\}$. In fact, if $M_i \parr N_i = \varnothing$ for some $i \in \{1,\dots,n\}$, then we could consider a graph $H$ with $|V_H| = |V_G| - 1$ such that

$$\begin{aligned} H^{\perp}(\!|M_1,\dots,M_{i-1},M_{i+1},\dots,M_n|\!) &= G^{\perp}(\!|M_1,\dots,M_{i-1},\varnothing,M_{i+1},\dots,M_n|\!)\\ H(\!|N_1,\dots,N_{i-1},N_{i+1},\dots,N_n|\!) &= G(\!|N_1,\dots,N_{i-1},\varnothing,N_{i+1},\dots,N_n|\!)\end{aligned}$$

satisfying the condition. Then, by Theorem 5, we can write $G = P(\!|G_1,\dots,G_m|\!)$ for a prime web $P$ and webs $G_1,\dots,G_m$. We can, without loss of generality, write

$$G(\!|N_1,\dots,N_n|\!) = P(\!|G_1(\!|N_1,\dots N_{k_1}|\!),\dots,G_m(\!|N_{k_{m-1}+1},\dots N_n|\!)|\!)$$

- If $G_i^\perp(\!|M_{k_{i-1}+1},\dots,M_{k_i}|\!) = \varnothing$ for some $i \in \{1,\dots,m\}$, then we can assume w.l.o.g. that $i = 1$. Then $N_1,\dots,N_{k_1}$ must be non-empty since we are assuming $M_j \parr N_j \neq \varnothing$ for all $j \in \{1,\dots,n\}$. Hence we can apply $\mathsf{s}_\otimes$ as follows and conclude by induction hypothesis:

$$\frac{\begin{array}{c}\mathbb{C}_1(\!|N_1,\dots,N_{k_1}|\!)\\ \| \, IH\\ G_1(\!|N_1,\dots,N_{k_1}|\!)\end{array} \otimes \begin{array}{c} \begin{array}{c}\mathbb{C}_k(\!|M_{k_1+1}\parr N_{k_1+1},\dots,M_{k_2}\parr N_{k_2}|\!)\\ \| \, IH\\ G_1^\perp(\!|M_{k_1+1},\dots,M_{k_2}|\!)\parr G_1(\!|N_{k_1+1},\dots,N_{k_2}|\!)\end{array} \otimes\cdots\otimes \begin{array}{c}\mathbb{C}_k(\!|M_{k_{n-1}+1}\parr N_{k_{n-1}+1},\dots,M_n\parr N_n|\!)\\ \| \, IH\\ G_1^\perp(\!|M_{k_{n-1}+1},\dots,M_n|\!)\parr G_1(\!|N_{k_{n-1}+1},\dots,N_n|\!)\end{array}\\ \| \, IH\\ \begin{array}{c}H^\perp(\!|G_2^\perp(\!|M_{k_1+1},\dots M_{k_2}|\!),\dots,G_m^\perp(\!|M_{k_{m-1}+1},\dots M_n|\!)|\!)\\ \hdashline P^\perp(\!|\varnothing,G_2^\perp(\!|M_{k_1+1},\dots,M_{k_2}|\!),\dots,G_m^\perp(\!|M_{k_{m-1}+1},\dots M_n|\!)|\!)\end{array} \parr \begin{array}{c}H(\!|G_2(\!|N_{k_1+1},\dots,N_{k_2}|\!),\dots,G_m(\!|N_{k_{m-1}+1},\dots,N_n|\!)|\!)\\ \hdashline P(\!|\varnothing,G_2(\!|N_{k_1+1},\dots,N_{k_2}|\!),\dots,G_m(\!|N_{k_{m-1}+1},\dots,N_n|\!)|\!)\end{array}\end{array}}{P^\perp(\!|\varnothing,G_2^\perp(\!|M_{k_1+1},\dots M_{k_2}|\!),\dots,G_m^\perp(\!|M_{k_{m-1}+1},\dots M_n|\!)|\!) \parr P(\!|G_1(\!|N_1,\dots N_{k_1}|\!),G_2(\!|N_{k_1+1},\dots N_{k_2}|\!),\dots,G_m(\!|N_{k_{m-1}+1},\dots N_n|\!)|\!)}\;2{\times}\mathsf{s}_\otimes$$

- If $G_i^\perp(\!|M_{k_{i-1}+1},\dots,M_{k_i}|\!) \neq \varnothing$ for all $i \in \{1,\dots,m\}$, and $\overset{P}{\curvearrowright} = \varnothing$ then we can apply $\mathsf{p}{\downarrow}$ as follows

$$\frac{\otimes_m\left(\begin{array}{c}\mathbb{C}_1(\!|M_1\parr N_1,\dots,M_{k_1}\parr N_{k_1}|\!)\\ \| \, IH\\ G_1^\perp(\!|M_1,\dots,M_{k_1}|\!)\parr G_1(\!|N_1,\dots,N_{k_1}|\!)\end{array},\dots,\begin{array}{c}\mathbb{C}_m(\!|M_{k_{m-1}+1}\parr N_{k_{m-1}+1},\dots,M_n\parr N_n|\!)\\ \| \, IH\\ G_m^\perp(\!|M_{k_{m-1}+1},\dots,M_n|\!)\parr G_m(\!|N_{k_{m-1}+1},\dots,N_n|\!)\end{array}\right)}{P^\perp(\!|G_1^\perp(\!|M_1,\dots,M_{k_1}|\!),\dots,G_m^\perp(\!|M_{k_{m-1}+1},\dots,M_n|\!)|\!)\parr P(\!|G_1(\!|N_1,\dots,N_{k_1}|\!),\dots,G_m(\!|N_{k_{m-1}+1},\dots,N_n|\!)|\!)}\;\mathsf{p}{\downarrow}$$

  where $\otimes_m$ is the complete undirected graph with $m$ vertices, and conclude by induction hypothesis.

- If $\overset{P}{\curvearrowright} \neq \varnothing$, we conclude similarly to the previous case by using $\mathsf{q}{\downarrow}$ instead of $\mathsf{p}{\downarrow}$ and therefore replacing the $\otimes_m$ with $\mathbb{C} = P$ (if $\overset{P}{\smile} = \varnothing$) or $\mathbb{C} = P^\perp$ (if $\overset{P}{\frown} = \varnothing$). ◀

▶ **Lemma 25.** *For every* $\mathsf{r}{\downarrow} \in \mathsf{GV}$ *we have that* $\mathsf{r}{\uparrow}$ *is derivable in* $\mathsf{GV} \cup \{\mathsf{i}{\uparrow}\}$.

**Proof.** This follows in exactly the same way as in any other deep inference system [21]. ◀

## 5 Cut elimination

The $\mathsf{i}{\uparrow}$ rule in (10) is the deep inference equivalent to the cut rule. We show in this section, that this rule is admissible for our systems, i.e., every web provable with cut is also provable without.

▶ **Theorem 26.** *The rule* $\mathsf{i}{\uparrow}$ *is admissible for* $\mathsf{GV}$ *and for* $\mathsf{GV}^{\mathsf{sl}}$.

The main consequence of this theorem is the transitivity of the consequence relation defined by implication $A \multimap B = A^\perp \parr B$ is transitive.

▶ **Corollary 27.** *Let $A$, $B$, and $C$ be any webs. If $A \multimap B$ and $B \multimap C$ are provable in* $\mathsf{GV}$ *(resp.* $\mathsf{GV}^{\mathsf{sl}}$*), then so is $A \multimap C$.*

**Proof.** Given proofs of $A^\perp \parr B$ and $B^\perp \parr C$, we also have a proof of $(A^\perp \parr B) \otimes (B^\perp \parr C)$. Applying $\mathsf{s}_\parr$ twice gives us $A^\perp \parr (B \otimes B^\perp) \parr C$, and then we can obtain $A^\perp \parr C$, via $\mathsf{i}{\uparrow}$. ◀

To prove Theorem 26, we rely on methods developed for deep inference in general [38, 21, 19, 4] and for graphical systems in particular [1]. There are two steps:

$$\vdash_{\mathsf{GX}\cup\{\mathsf{i}\uparrow\}} G \iff \vdash_{\mathsf{GX}\cup\{\mathsf{ai}\uparrow,\mathsf{p}\uparrow,\mathsf{q}\uparrow\}} G \iff \vdash_{\mathsf{GX}} G \tag{11}$$

where $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$ and $G$ is an arbitrary web. The first step is decomposing the general $\mathsf{i}\uparrow$ (which introduces an arbitrary web when going up in a proof) into smaller steps that are easier to control, and it is achieved with Lemma 25 and the following proposition, that is an immediate consequence of Lemma 24.

▶ **Proposition 28.** *The rule* $\mathsf{i}\uparrow$ *can be derived with* $\{\mathsf{ai}\uparrow, \mathsf{p}\uparrow, \mathsf{q}\uparrow\}$. *And dually,* $\mathsf{i}\downarrow$ *can be derived with* $\{\mathsf{ai}\downarrow, \mathsf{p}\downarrow, \mathsf{q}\downarrow\}$.

It remains to show the second step in (11), i.e., the following theorem.

▶ **Theorem 29.** *The rules* $\mathsf{ai}\uparrow$, $\mathsf{p}\uparrow$, *and* $\mathsf{q}\uparrow$ *are admissible for* $\mathsf{GV}$ *and* $\mathsf{GV^{sl}}$.

Proving this theorem follows the same outline as in other deep inference systems, via *splitting* and *context reduction* stated below.

▶ **Lemma 30** (Tensor Splitting). *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$, *let* $A$ *and* $B$ *be nonempty webs, and let* $G$ *be an arbitrary web. If* $\vdash_{\mathsf{GX}} G \parr (A \otimes B)$, *then there are webs* $K_A$ *and* $K_B$ *such that there are derivations*

$$\begin{array}{c} K_A \parr K_B \\ \mathcal{D}_G \,\Vert\, \mathsf{GX} \\ G \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_A \,\Vert\, \mathsf{GX} \\ K_A \parr A \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_B \,\Vert\, \mathsf{GX} \\ K_B \parr B \end{array} \quad .$$

Note that the well-known splitting tensor lemma for linear logic proof-nets [10] states the existence of a splitting tensor, whereas Lemma 30 (and any splitting tensor lemma in deep inference in general) says that every tensor can be made splitting by first reducing the context.

▶ **Lemma 31** (Atomic Splitting). *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$ *and* $G$ *be a web. If* $\vdash_{\mathsf{GX}} G \parr a$ *for an atom* $a$, *then there is a derivation*

$$\begin{array}{c} a^{\perp} \\ \mathcal{D}_G \,\Vert\, \mathsf{GX} \\ G \end{array} \quad .$$

▶ **Lemma 32** (Context Reduction). *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$, *Let* $A$ *be a web and* $\mathcal{C}[\Box]$ *be a context, such that* $\vdash_{\mathsf{GX}} \mathcal{C}[A]$. *Then there is a web* $K$, *such that for any web* $\mathcal{X}$ *there are derivations*

$$\begin{array}{c} K \parr \mathcal{X} \\ \mathcal{D}_{\mathcal{X}} \,\Vert\, \mathsf{GX} \\ \mathcal{C}[\mathcal{X}] \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_A \,\Vert\, \mathsf{GX} \\ K \parr A \end{array}$$

These three lemmas are already enough to prove the admissibility of $\mathsf{ai}\uparrow$ for $\mathsf{GV}$ and $\mathsf{GV^{sl}}$: Assume we have a proof of $\mathcal{C}[a^{\perp} \otimes a]$. By the three lemmas above, we get webs $K$, $K_{a^{\perp}}$, $K_a$ and derivations

$$\begin{array}{c} K \parr \varnothing \\ \mathcal{D}_{\varnothing} \,\Vert\, \mathsf{GX} \\ \mathcal{C}[\varnothing] \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_K \,\Vert\, \mathsf{GX} \\ K \parr (a^{\perp} \otimes a) \end{array} \quad \begin{array}{c} K_{a^{\perp}} \parr K_a \\ \mathcal{D}_K \,\Vert\, \mathsf{GX} \\ K \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_{a^{\perp}} \,\Vert\, \mathsf{GX} \\ K_{a^{\perp}} \parr a^{\perp} \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_a \,\Vert\, \mathsf{GX} \\ K_a \parr a \end{array} \quad \begin{array}{c} a \\ \mathcal{D}_1 \,\Vert\, \mathsf{GX} \\ K_{a^{\perp}} \end{array} \quad \begin{array}{c} a^{\perp} \\ \mathcal{D}_2 \,\Vert\, \mathsf{GX} \\ K_a \end{array}$$

With the derivations $\mathcal{D}_{\varnothing}$, $\mathcal{D}_K$, $\mathcal{D}_1$, $\mathcal{D}_2$ and an instance of $\mathsf{ai}\downarrow$, we can now obtain a proof of $\mathcal{C}[\varnothing]$, as desired.

In order to prove admissibility of $\mathsf{p}\!\uparrow$ and of $\mathsf{q}\!\uparrow$ in a similar way, we need two more variants of the splitting lemma. The first (Lemma 33) deals with prime webs in general, and the second (Lemma 34) deals with complete webs, i.e., webs in which for any two vertices $x, y$, we either have $x \frown y$ or $x \curvearrowright y$ or $x \curvearrowleft y$.

▶ **Lemma 33** (Prime Web Splitting). *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$*, let* $P \neq \parr$ *be a prime web with* $|V_P| = n$*, let* $M_1, \ldots, M_n$ *be non-empty webs, and let* $G$ *be an arbitrary web. If* $\vdash_{\mathsf{GX}} G \parr P(\!|M_1, \ldots, M_n|\!)$*, then one of the following holds:*

**(A)** *there are webs* $K_1, \ldots, K_n$*, such that there are derivations*

$$\begin{array}{c} P^\perp(\!|K_1, \ldots, K_n|\!) \\ \mathcal{D}_G \,\|\, \mathsf{GX} \\ G \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_i \,\|\, \mathsf{GX} \\ K_i \parr M_i \end{array} \qquad \textit{for all } i \in \{1, \ldots, n\};$$

**(B)** *there are webs* $K_X$ *and* $K_Y$*, such that there are derivations*

$$\begin{array}{c} K_X \parr K_Y \\ \mathcal{D}_G \,\|\, \mathsf{GX} \\ G \end{array} , \quad \begin{array}{c} \varnothing \\ \mathcal{D}_X \,\|\, \mathsf{GX} \\ K_X \parr M_i \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_Y \,\|\, \mathsf{GX} \\ K_Y \parr P(\!|M_1, \ldots, M_{i-1}, \varnothing, M_{i+1}, \ldots, M_n|\!) \end{array} \quad \textit{for some } i \in \{1, \ldots, n\};$$

**(C)** *only if* $\mathsf{GX} = \mathsf{GV^{sl}}$ *and* $\overset{P}{\frown} = \varnothing$*: there are webs* $K_X$ *and* $K_Y$*, and some* $k \in \{2, \ldots, n-1\}$ *such that, w.l.o.g.,* $y \not\curvearrowright x$ *for all* $x \in \bigcup_{i=1}^{k} M_i$ *and* $y \in \bigcup_{j=k+1}^{n} M_j$*, and there are derivations*

$$\begin{array}{c} K_X \lhd K_Y \\ \mathcal{D}_G \,\|\, \mathsf{GX} \\ G \end{array} , \quad \begin{array}{c} \varnothing \\ \mathcal{D}_X \,\|\, \mathsf{GX} \\ K_X \parr P(\!|M_1, \ldots, M_k, \varnothing, \ldots, \varnothing|\!) \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_Y \,\|\, \mathsf{GX} \\ K_Y \parr P(\!|\varnothing, \ldots, \varnothing, M_{k+1}, \ldots, M_n|\!) \end{array}$$

▶ **Lemma 34** (Complete Web Splitting). *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV^{sl}}\}$*. Let* $\mathbb{C}$ *be a complete web with* $|V_{\mathbb{C}}| = n \geq 2$*, let* $M_1, \ldots, M_n$ *be non-empty webs, and let* $G$ *be an arbitrary web. If* $\vdash_{\mathsf{GV}} G \parr \mathbb{C}(\!|M_1, \ldots, M_n|\!)$*, then there are webs* $K_1, \ldots, K_n$ *such that there are derivations*

$$\begin{array}{c} \mathbb{C}^\perp(\!|K_1, \ldots, K_n|\!) \\ \mathcal{D}_G \,\|\, \mathsf{GX} \\ G \end{array} \quad \textit{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_i \,\|\, \mathsf{GX} \\ K_i \parr M_i \end{array} \qquad \textit{for all } i \in \{1, \ldots, n\}.$$

These lemmas are now sufficient to complete the proof of rule elimination (Theorem 29).

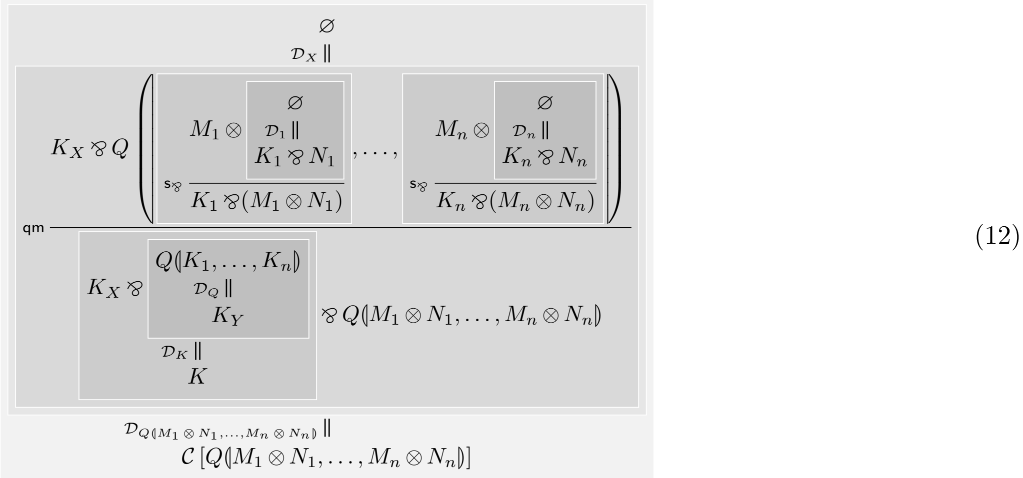
**Proof of Theorem 29.** For $\mathsf{ai}\!\uparrow$, this has been shown above. For $\mathsf{p}\!\uparrow$, we proceed similarly as in [1, 2] for $\mathsf{GS}$. For $\mathsf{q}\!\uparrow$, assume $\vdash_{\mathsf{GX}} \mathcal{C}\left[Q(\!|M_1, \ldots, M_n|\!) \otimes Q^\perp(\!|N_1, \ldots, N_n|\!)\right]$. By Lemma 32 there is a web $K$, such that for any web $\mathcal{X}$ there are the derivations

$$\begin{array}{c} K \parr \mathcal{X} \\ \mathcal{D}_{\mathcal{X}} \,\|\, \mathsf{GX} \\ \mathcal{C}[\mathcal{X}] \end{array} \quad \text{and} \quad \begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ K \parr Q(\!|M_1, \ldots, M_n|\!) \otimes Q^\perp(\!|N_1, \ldots, N_n|\!) \end{array}$$

By Lemma 30 we have webs $K_X$ and $K_Y$ and the derivations below left

$$\begin{array}{c} K_X \parr K_Y \\ \mathcal{D}_K \,\|\, \mathsf{GX} \\ K \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_X \,\|\, \mathsf{GX} \\ K_X \parr Q(\!|M_1, \ldots, M_n|\!) \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_Y \,\|\, \mathsf{GX} \\ K_Y \parr Q^\perp(\!|N_1, \ldots, N_n|\!) \end{array} \quad \Bigg| \quad \begin{array}{c} Q(\!|K_1, \ldots, K_n|\!) \\ \mathcal{D}_Q \,\|\, \mathsf{GX} \\ K_Y \end{array} \quad \begin{array}{c} \varnothing \\ \mathcal{D}_i \,\|\, \mathsf{GX} \\ K_i \parr N_i \end{array}$$

Since $\overset{Q}{\frown} = \varnothing$, we have that $Q^\perp$ is a complete web, and we can apply Lemma 34 to $\mathcal{D}_Y$ giving us $K_1, \ldots, K_n$ and the derivations above right (for all $i \in \{1, \ldots, n\}$). We conclude with



(12)

◀

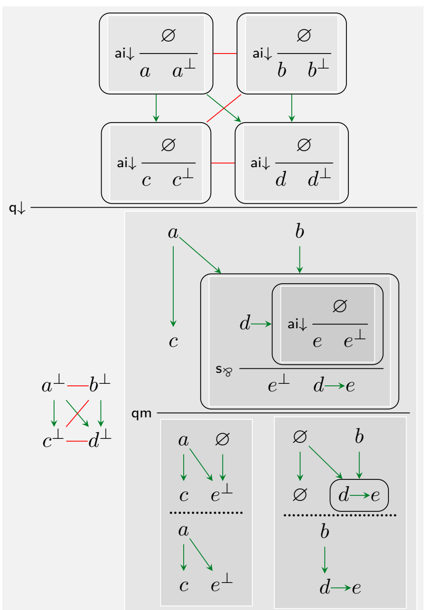
Observe that Lemma 30 is just a special case of Lemma 34 and also a special case of Lemma 33, as cases (A) and (B) of Lemma 33 collapse both to Lemma 30 if $P = \otimes$. Therefore, it only remains to prove Lemma 31, Lemma 32, Lemma 33, and Lemma 34, which is done by induction on the size of the web to be proved.

▶ **Definition 35.** *We define the **size** of a web $G$ as the pair $\|G\| = \langle |V_G|, |\curvearrowright| + 2|\smile| \rangle$, ordered lexicographically, i.e., if $\|G\| = \langle n_G, m_G \rangle$ and $\|H\| = \langle n_H, m_H \rangle$, then we have $G < H$ if $n_G < n_H$ or if $n_G = n_H$ and $m_G < m_H$.*

Even though the general pattern of the overall argument to prove Theorem 26 is similar to other deep inference systems, the details are much more involved. Some of the problems have already been observed for the system GS in [1]. For example, Lemma 32, Lemma 33 and Lemma 34 are not proved one after the other, but simultaneously, by mutual induction on the size (defined above) of the conclusion. An additional difficulty for GV and $\mathsf{GV}^{\mathsf{sl}}$ is the explosion of the cases to consider. When proving Lemma 33, we have to do a case analysis on the last rule applied to the proof of $G \parr P(\!|M_1, \ldots, M_n|\!)$. More details on the splitting proof can be found in Appendix A.

▶ Remark 36. The $\mathsf{s}_\otimes$ is usually admissible in a deep inference system, as it is an *up-rule* (and dual to $\mathsf{s}_\parr$). This was also the case for GS [1]. However, when we consider mixed graphs, the rule is no longer admissible. Or, put differently, cut elimination does not hold if we remove the rule from the system (see Appendix B).

▶ **Example 37.** To illustrate the dynamics of splitting, consider the alternative proofs in Figure 5 which prove the same web as found in the conclusion of the proof on the left of Figure 4. The proofs in Figure 5 can be obtained by applying splitting to the first proof in Figure 4. For the proof on the left of Figure 5, apply Lemma 34 to the leftmost disjoint graph, which ensures the existence of the derivation found below the instance of the $\mathsf{q}{\downarrow}$ rule, that does not change the sub-graph selected. Similarly, the second proof in Figure 5 is obtained by applying splitting to the connected sub-graph labelled with $a$, $c$ and $e^\perp$.



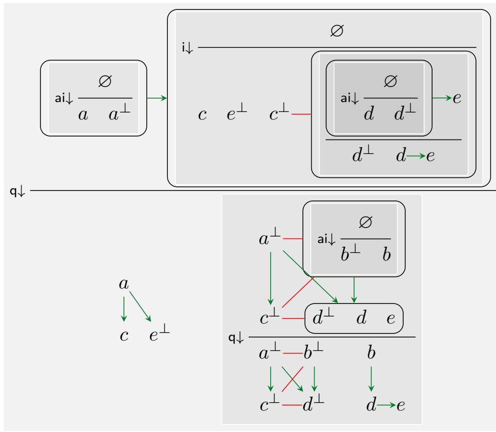


**Figure 5** Alternative proofs of the web proved by the left derivation in Figure 4.

## 6 Analyticity, Conservativity, Complexity

In this section, we discuss some consequences of the cut-elimination result obtained in the previous section. The most important is that GV and $\mathsf{GV}^{\mathsf{sl}}$ are both analytic. Since deep inference systems in general, and our graphical proof systems in particular, do not have a "subformula property" in the traditional sense, we present here a formal notion of what we mean by analyticity of a graphical proof system. For this, we start from the notion of *connector* given in Definition 8.

▶ **Definition 38.** *A* ***subconnector*** *of $G$ is a prime web that is an induced subgraph of a connector in $G$. A connector (or subconnector) $P$ is called* ***proper*** *if $\overset{P}{\frown} \neq \varnothing$. A proof $\mathcal{D}$ of $G$ is* ***analytic*** *if every proper connector of a web in $\mathcal{D}$ is a subconnector of $G$.*

▶ **Theorem 39** (Analyticity). *If $\vdash_{\mathsf{GX}} G$ for $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV}^{\mathsf{sl}}\}$, then $G$ admits an analytic proof in* GX*.*

**Proof Sketch.** This is proved similarly to the same result for GS given in [2], by taking into account the additional rules in GV and $\mathsf{GV}^{\mathsf{sl}}$. When going bottom-up in a derivation (as in proof search), the only two rules that can introduce a new proper connector that is not a subconnector of the conclusion are $\mathsf{s}_{\otimes}$ (when the $N$ in Figure 2 is empty) and $\mathsf{q}\downarrow$ (when $L_i$ in Figure 2 is empty for some $i \in \{1, \dots, |V_Q|\}$). Since the premise of every proof is $\varnothing$, all these connectors have to be destroyed eventually. This can happen via the rules $\mathsf{ai}\downarrow$ or $\mathsf{s}_{\otimes}$ (one immediate submodule of the connector is removed), or via the rule $\mathsf{p}\downarrow$ or $\mathsf{q}\downarrow$ (the connector itself is destroyed, the submodules remain). It can now be shown via rule permutations that the rule instance that creates the connector and the rule instance that destroys it can be brought together, such that we can rearrange the derivation in a way that the offending connector does not occur. ◀

▶ **Corollary 40.** *Let* $\mathsf{GX} \in \{\mathsf{GV}, \mathsf{GV}^{\mathsf{sl}}\}$ *and let* $G$ *be an undirected graph. If* $\vdash_{\mathsf{GX}} G$*, then* $G$ *admits a derivation* $\mathcal{D}$ *in* GX *such that every connector of a web in* $\mathcal{D}$ *is a subconnector of* $G$*.*

**Proof Sketch.** For proper connectors this has been shown in Theorem 39. Hence we only need to consider $P$-connectors that are directed prime graphs. The only way to create such a new $P$-connector is via $\mathsf{s}_{⅋}$, $\mathsf{q}{\downarrow}$,or $\mathsf{qm}$. Then $P$ is either destroyed by destroying all its modules, or it is destroyed after merging via $\mathsf{qm}$ (or $\mathsf{q}{\downarrow}$ if $P = \lhd$) with another $P$-connector. In both cases we use a similar rule permutation argument as in the proof of Theorem 39 to eliminate the occurrence of the $P$-connector. ◀

As a consequence of Theorem 39, we know that, in an analytic proof, any proper connector occurring in the conclusion of a $\mathsf{q}{\downarrow}$ also occurs in its premise. This condition is not required by definition for the rule $\mathsf{q}{\downarrow}$ (see Figure 2) where the side condition only asks that $L_i ⅋ J_i$ is non-empty: the rule might introduce bottom-up a new connector whenever a $L_i$ is empty. The inclusion of the additional restriction $L_i \neq \varnothing$ for all $i \in \{1, \ldots, n\}$ for $\mathsf{q}{\downarrow}$ in the definition of the rule would require a more complex formulation of the splitting statement.

To better understand that subtlety consider the example in Figure 6. The proof on the left is obtained by applying splitting to the leftmost connected component of the conclusion; while the proof on the right has a shape obtained by applying splitting to the rightmost connected component. In the latter proof, the bottommost $\mathsf{q}{\downarrow}$ introduces a new connector establishing relations between the modules of the leftmost and central components of the conclusion. As indicated in the figure, this is achieved by introducing empty modules in different places in both webs affected by that instance of the $\mathsf{q}{\downarrow}$ rule, making use of the weak side condition mentioned above. The application of a restricted version of the $\mathsf{q}{\downarrow}$ would force the merge of the two webs to establish additional relations which later would prevent a successful proof. However, observe that this proof violates analyticity.

In contrast, the proof on the left is analytic. Even though that proof also appeals to the same weak side conditions and introduces in the premise of the qm-rule instance a new connector that is not a subconnector of the conclusion, this new connector is not proper because it contains no $\frown$-edge. This is the reason for restricting our analyticity result to proper connectors.

Nonetheless, this restricted form of analyticity is enough to show that both, GV and $\mathsf{GV}^{\mathsf{sl}}$, are conservative extensions of GS and of BV.

▶ **Theorem 41** (Conservativity). *Let* $G$ *be an undirected graph. Then* $\vdash_{\mathsf{GS}} G$ *iff* $\vdash_{\mathsf{GV}} G$ *iff* $\vdash_{\mathsf{GV}^{\mathsf{sl}}} G$*.*

**Proof.** Assume we have a derivation of $G$ in GV or $\mathsf{GV}^{\mathsf{sl}}$. By Corollary 40, there is also a derivation in which all occurring webs are also undirected graphs. Hence all occurring rules are $\mathsf{ai}{\downarrow}$, $\mathsf{s}_{⅋}$, $\mathsf{s}_{\otimes}$, and $\mathsf{p}{\downarrow}$. Now the result follows from admissibility of $\mathsf{s}_{\otimes}$ for GS (see Theorem 23). ◀

▶ **Theorem 42** (Conservativity). *Let* $G$ *be a relation web. Then* $\vdash_{\mathsf{BV}} G$ *iff* $\vdash_{\mathsf{GV}} G$ *iff* $\vdash_{\mathsf{GV}^{\mathsf{sl}}} G$*.*

**Proof.** In Lemma 21, we have shown that every proof of BV is also a proof in GV, and therefore also in $\mathsf{GV}^{\mathsf{sl}}$. Conversely, assume we have a proof $\mathcal{D}$ of $G$ in GV or $\mathsf{GV}^{\mathsf{sl}}$. Since $G$ is a relation web, all its connectors are in $\{\otimes, \lhd, ⅋\}$. Hence, by Corollary 40, there is a derivation $\tilde{\mathcal{D}}$ in which each occurring web is a relation web. Therefore, each instance of $\mathsf{p}{\downarrow}$ can be simulated by two instances of $\mathsf{s}$; each $\mathsf{s}_{⅋}$ is an instance of $\mathsf{s}$ or $\mathsf{q}_{\mathsf{BV}}{\downarrow}$; each instance of $\mathsf{s}_{\otimes}$ is an instance of $\mathsf{s}$ or $\mathsf{q}_{\mathsf{BV}}{\uparrow}$; each instance of $\mathsf{q}{\downarrow}$ or $\mathsf{qm}$ or $\mathsf{sl}$ is an instance of $\mathsf{q}_{\mathsf{BV}}{\downarrow}$, and each instance of $\mathsf{q}{\uparrow}$ is an instance of $\mathsf{q}_{\mathsf{BV}}{\uparrow}$. Now the result follows from the fact that $\mathsf{q}_{\mathsf{BV}}{\uparrow}$ is admissible for BV (see Theorem 22). ◀

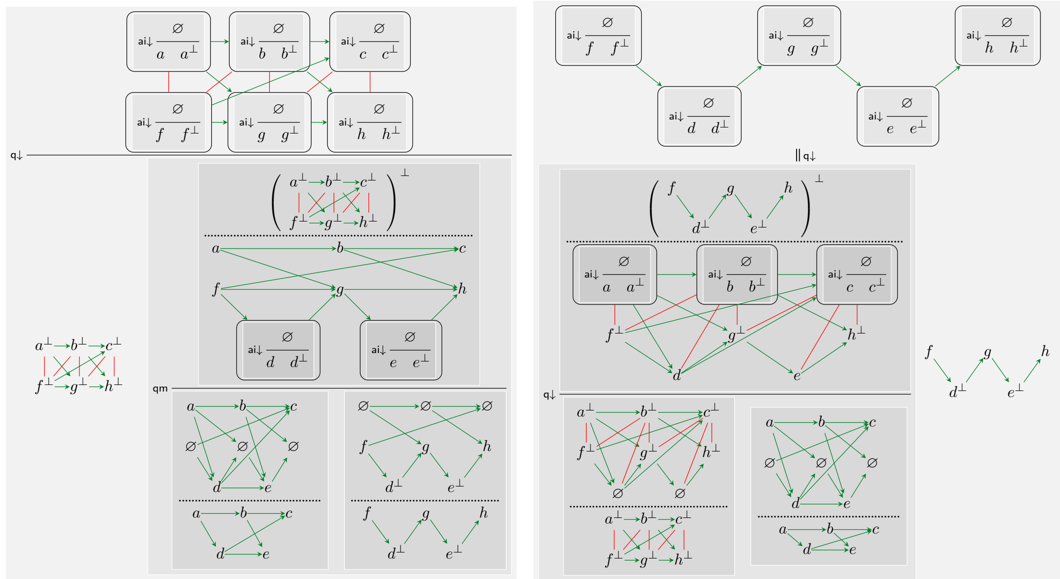


**Figure 6** Two proofs of the same web. Only the derivation on the left is analytic: the qm in the derivation on the left introduces a new connector, which is not proper; the bottommost q↓ in the derivation on the right introduces a new proper connector, violating analyticity.

▶ **Corollary 43.** *Provability in* GV *and in* $\mathsf{GV}^{\mathsf{sl}}$ *is* **NP***-complete.*

**Proof.** It has been shown before that provability in BV is **NP**-complete [28]. Hence, by Theorem 42 we obtain **NP**-hardness of provability in GV and $\mathsf{GV}^{\mathsf{sl}}$. For containment in **NP**, observe that for any rule instance $\mathsf{r}\,\frac{H}{G}$ in $\mathsf{GV}^{\mathsf{sl}}$ we have that $\|H\| < \|G\|$. For a web $G$ we have $|\overset{G}{\curvearrowright}| + 2|\overset{G}{\smile}| < 2(|V_G|^2)$. Therefore any derivation in $\mathsf{GV}^{\mathsf{sl}}$ (or in GV) of a web $G$ has at most length $\mathcal{O}(|V_G|^3)$. ◀

## 7 Conclusion

We presented two analytic proof systems, GV and $\mathsf{GV}^{\mathsf{sl}}$ (Figure 2), that conservatively extend the logic BV to graphs (Theorem 42). The graphs, called webs (Definition 12), feature both undirected and directed edges, which generalise, respectively, the multiplicative conjunction of linear logic, and the non-commutative connective *seq* of BV. In addition, our systems are conservative extensions of GS (Theorem 41), which features general simple graphs. This is no coincidence, since GS was conceived precisely as the minimal core of graphical logics like GV for which it was already clear that we needed to generalise the tools of proof theory. Indeed, we were able to adapt the deep inference technology developed for GS to prove cut elimination for GV and $\mathsf{GV}^{\mathsf{sl}}$ (Theorem 26 and Section 5).

Differences compared to GS surprised us. Notably, the $\mathsf{s}_\otimes$ rule is not admissible in GV (explained further in Appendix B), which in fact simplifies splitting (Lemmas 30-34), since $\mathsf{s}_\otimes$ can be used to remove contexts (Lemma 32). A nuance of GV is that rules for directed graphs require weaker side-conditions than similar rules for symmetric graphs. For example, if we restrict q↓ and qm such that all modules in one of the graphs are non-empty, then there is no proof of the web in Fig. 6 – an observation we can use to prove that such a restricted system cannot satisfy cut elimination. This forced us to take additional care defining analyticity Theorem 39, used to establish conservativity.

We draw attention to an open problem that we found more challenging than expected. Ideally, we would like to have an extension of GV admitting the ***homomorphism rule*** below on the left, making it possible to prove additional graphs beyond the scope of the systems proposed in this paper.

$$\mathsf{h}{\downarrow}\ \frac{H(\!| M_1, \ldots, M_n |\!)}{G(\!| M_1, \ldots, M_n |\!)}\ V_G = V_H,\ \overset{H}{\frown} = \overset{G}{\frown} = \varnothing,\ \overset{H}{\curvearrowright} \supset \overset{G}{\curvearrowright}$$

$$\begin{array}{cccc} \mathtt{e}_a \to \mathtt{e}_b \to \mathtt{e}_c \to \mathtt{e}_d \\ \mathtt{d}_a \to \mathtt{d}_b \to \mathtt{d}_c \to \mathtt{d}_d \end{array} \quad \multimap \quad \begin{array}{cccc} \mathtt{e}_a \to \mathtt{e}_b \to \mathtt{e}_c \to \mathtt{e}_d \\ \mathtt{d}_a \to \mathtt{d}_b \to \mathtt{d}_c \to \mathtt{d}_d \end{array}$$

By means of example, the implication $\mathtt{Q}_3 \multimap \mathtt{Q}_4$ above on the right, internalising in the graphical logic the fact that a 4-queue can simulate the behaviour of a 3-queue, cannot be proven in $\mathsf{GV}^{\mathsf{sl}}$. The challenge we encounter in handling graph homomorphisms is due to its wild behaviour on graph modular decomposition. In fact, the rules $\mathsf{q}_{\mathsf{BV}}{\downarrow}$, $\mathsf{qm}$ and $\mathsf{sl}$ are special instances of this homomorphism rule with a controlled behaviour with respect to modular decomposition. Further insight is required to prove cut elimination for extensions of GV where the rule $\mathsf{h}{\downarrow}$ is admissible. The system $\mathsf{GV}^{\mathsf{sl}}$, which demanded a dedicated case in splitting (Lemma 33) to handle $\mathsf{sl}$, is a first step towards that aim.

## References


1. Matteo Acclavio, Ross Horne, and Lutz Straßburger. Logic beyond formulas: A proof system on graphs. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 38–52, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3373718.3394763`.
2. Matteo Acclavio, Ross Horne, and Lutz Straßburger. An analytic propositional proof system on graphs. *Logical Methods in Computer Science*, 2022. to appear. `doi:10.48550/arXiv.2012.01102`.
3. Matteo Acclavio and Roberto Maieli. Generalized connectives for multiplicative linear logic. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*, volume 152 of *LIPIcs*, pages 6:1–6:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2020.6`.
4. Andrea Aler Tubella. *A study of normalisation through subatomic logic*. PhD thesis, University of Bath, 2017.
5. Andrea Aler Tubella and Lutz Straßburger. Introduction to deep inference. Lecture, August 2019. URL: `https://hal.inria.fr/hal-02390267`.
6. Denis Bechet, Philippe de Groote, and Christian Retoré. A complete axiomatisation of the inclusion of series-parallel partial orders. In H. Common, editor, *Rewriting Techniques and Applications, RTA 1997*, volume 1232 of *LNCS*, pages 230–240. Springer, 1997.
7. Carmen Bruckmann, Peter F. Stadler, and Marc Hellmuth. From modular decomposition trees to rooted median graphs. *Discrete Applied Mathematics*, 310:1–9, 2022. `doi:10.1016/j.dam.2021.12.017`.
8. Paola Bruscoli. A purely logical account of sequentiality in proof search. In Peter J. Stuckey, editor, *Logic Programming*, pages 302–316, Berlin, Heidelberg, 2002. Springer. `doi:10.1007/3-540-45619-8_21`.
9. Cameron Calk, Anupam Das, and Tim Waring. Beyond formulas-as-cographs: an extension of Boolean logic to arbitrary graphs, 2020. `doi:10.48550/arXiv.2004.12941`.
10. Vincent Danos and Laurent Regnier. The structure of the multiplicatives. *Arch. Math. Log.*, 28(3):181–203, 1989. `doi:10.1007/BF01622878`.
11. Yuxin Deng, Robert J. Simmons, and Iliano Cervesato. Relating reasoning methodologies in linear logic and process algebra. *Mathematical Structures in Computer Science*, 26(5):868–906, 2016. `doi:10.1017/S0960129514000413`.

12 Pierre-Malo Deniélou and Nobuko Yoshida. Buffered communication analysis in distributed multiparty sessions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 343–357, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15375-4_24.

13 A. Ehrenfeucht, T. Harju, and G Rozenberg. *The Theory of 2-Structures A Framework for Decomposition and Transformation of Graphs.* World Scientific, 1999. doi:10.1142/4197.

14 Uli Fahrenberg, Christian Johansen, Georg Struth, and Ratan Bahadur Thapa. Generating posets beyond $n$. In Uli Fahrenberg, Peter Jipsen, and Michael Winter, editors, *Relational and Algebraic Methods in Computer Science*, pages 82–99, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-43520-2_6.

15 Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004. doi:10.1145/988672.988756.

16 Tibor Gallai. Transitiv orientierbare Graphen. *Acta Mathematica Academiae Scientiarum Hungarica*, 18(1–2):25–66, 1967.

17 Jean-Yves Girard. Multiplicatives. In Gabriele Lolli, editor, *Logic and Computer Science: New Trends and Applications*, pages 11–34. Rosenberg & Sellier, 1987.

18 Jay L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61:199–224, 1988. doi:10.1016/0304-3975(88)90124-7.

19 Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007. doi:10.1145/1182613.1182614.

20 Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A proof calculus which reduces syntactic bureaucracy. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *LIPIcs*, pages 135–150, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.RTA.2010.135.

21 Alessio Guglielmi and Lutz Straßburger. Non-commutativity and MELL in the calculus of structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 54–68, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-44802-0_5.

22 Michel Habib and Christophe Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010. doi:10.1016/j.cosrev.2010.01.001.

23 Ross Horne. Session subtyping and multiparty compatibility using circular sequents. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2020.12.

24 Ross Horne, Sjouke Mauw, and Alwen Tiu. Semantics for specialising attack trees based on linear logic. *Fundam. Inform.*, 153(1-2):57–86, 2017. doi:10.3233/FI-2017-1531.

25 Ross Horne and Alwen Tiu. Constructing weak simulations from linear implications for processes with private names. *Mathematical Structures in Computer Science*, 29(8):1275–1308, 2019. doi:10.1017/S0960129518000452.

26 Ross Horne, Alwen Tiu, Bogdan Aman, and Gabriel Ciobanu. De Morgan dual nominal quantifiers modelling private names in non-commutative logic. *ACM Trans. Comput. Log.*, 20(4):22:1–22:44, 2019. doi:10.1145/3325821.

27 Lee O James, Ralph G Stanton, and Donald D Cowan. Graph decomposition for undirected graphs. In *Proceedings of the Third Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1972)*, pages 281–290, 1972.

28 Ozan Kahramanoğulları. System BV is NP-complete. *Annals of Pure and Applied Logic*, 152(1-3):107–121, 2008. doi:10.1016/j.apal.2007.11.005.

29 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

**30** László Lovász and Michael D Plummer. *Matching theory*, volume 367. American Mathematical Soc., 2009.

**31** Ross M. McConnell and Jeremy P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, pages 536–545, USA, 1994. Society for Industrial and Applied Mathematics.

**32** Lê Thành Dũng Nguyễn and Lutz Straßburger. BV and Pomset Logic are not the same. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, volume 216 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CSL.2022.3`.

**33** Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981. Special Issue Semantics of Concurrent Computation. `doi:10.1016/0304-3975(81)90112-2`.

**34** Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986. `doi:10.1007/BF01379149`.

**35** Christian Retoré. Perfect matchings and series-parallel graphs: multiplicative proof nets as R&B-graphs. *Electronic Notes in Theoretical Computer Science*, 3, 1996. `doi:10.1016/S1571-0661(05)80416-5`.

**36** Christian Retoré. Pomset logic: A non-commutative extension of classical linear logic. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 300–318, Berlin, Heidelberg, 1997. Springer. `doi:10.1007/3-540-62688-3_43`.

**37** Christian Retoré. Handsome proof-nets: perfect matchings and cographs. *Theoretical Computer Science*, 294(3):473–488, 2003. `doi:10.1016/S0304-3975(01)00175-X`.

**38** Lutz Straßburger. *Linear Logic and Noncommutativity in the Calculus of Structures*. PhD thesis, Technische Universität Dresden, 2003.

**39** Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. The recognition of series parallel digraphs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12. ACM, 1979. `doi:10.1137/0211023`.


# A Proof of Splitting and Context Reduction

**Sketch of Proof of Splitting Lemma for Prime Webs (Lemma 33).** The proof proceeds by case analysis of the last rule in a derivation of a graph $G \parr P(\!|M_1, \ldots, M_n|\!)$.

- the last rule $\mathsf{r}$ acts inside $G$ or any of the $M_i$,

$$\begin{array}{c}\varnothing\\ \mathcal{D}' \,\|\, \mathsf{GX}\\ \mathsf{r}\dfrac{G'}{G} \parr P(\!|M_1, \ldots, M_n|\!)\end{array} \quad \text{or} \quad \begin{array}{c}\varnothing\\ \mathcal{D}' \,\|\, \mathsf{GX}\\ G \parr P(\!|M_1, \ldots, M_{i-1}, \mathsf{r}\dfrac{M_i'}{M_i}, M_{i+1}, \ldots, M_n|\!)\end{array}$$

  then we can conclude by inductive hypothesis using a rules permutation argument.

- the last rule is a $\mathsf{s}_{\parr}$ moving a connected component of $G = G'' \parr G'$ inside $P$, that is,

$$\mathsf{s}_{\parr}\dfrac{\begin{array}{c}\varnothing\\ \mathcal{D}'' \,\|\, \mathsf{GX}\\ G'' \parr P(\!|M_1, \ldots, M_{i-1}, G' \parr M_i, M_{i+1} \ldots, M_n|\!)\end{array}}{G'' \parr G' \parr P(\!|M_1, \ldots, M_n|\!)} \quad \text{or} \quad \mathsf{s}_{\parr}\dfrac{\begin{array}{c}\varnothing\\ \mathcal{D}'' \,\|\, \mathsf{GX}\\ G'' \parr R(\!|M_1', \ldots, M_i', G', M_{i+1}', \ldots, M_n'|\!)\end{array}}{G'' \parr G' \parr \dfrac{R(\!|M_1', \ldots, M_i', \varnothing, M_{i+1}', \ldots, M_n'|\!)}{P(\!|M_1, \ldots, M_k|\!)}}$$

  and we prove the result by inductive hypothesis on the conclusion of $\mathcal{D}''$

- the last rule is a $\mathsf{s}_{\parr}$ moving the connected component $P(\!|M_1,\ldots,M_n|\!)$ inside $G$, that is,

$$\mathsf{s}_{\parr}\frac{\begin{array}{c}\varnothing\\ \mathcal{D}'\,\|\,\mathsf{GX}\\ G''\parr R(\!|N_1,\ldots,N_{i-1},P(\!|M_1,\ldots,M_n|\!)\parr N_i,N_{i+1},\ldots,N_m|\!)\end{array}}{G''\parr R(\!|N_1,\ldots,N_m|\!)\parr P(\!|M_1,\ldots,M_n|\!)}$$

  In this case, we conclude by inductive hypothesis using Lemma 32.

- the last rule is a $\mathsf{s}_{\otimes}$, leading to one of the following two cases:

$$\mathsf{s}_{\otimes}\frac{\begin{array}{c}\varnothing\\ \mathcal{D}''\,\|\,\mathsf{GX}\\ G\parr(M_i\otimes P(\!|M_1,\ldots,M_{i-1},\varnothing,M_{i+1},\ldots,M_n|\!))\end{array}}{G\parr P(\!|M_1,\ldots,M_n|\!)} \quad\text{or}\quad \mathsf{s}_{\otimes}\frac{\begin{array}{c}\varnothing\\ \mathcal{D}''\,\|\,\mathsf{GX}\\ G\parr(P(\!|M_1,\ldots,M_{i-1},M_i,M_{i+1}\ldots,M_n|\!)\otimes M'_i)\end{array}}{G\parr P(\!|M_1,\ldots,M_{i-1},M_i\otimes M'_i,M_{i+1}\ldots,M_n|\!)}$$

  Using Lemma 30 the first case immediately leads to Case (B), while the second requires the use of inductive hypothesis to conclude.

- the last rule is a $\mathsf{p}\!\downarrow$, leading to one of the two following cases:

$$\mathsf{p}\!\downarrow\frac{\begin{array}{c}\varnothing\\ \mathcal{D}'\,\|\,\mathsf{GX}\\ G''\parr((N_1\parr M_1)\otimes\cdots\otimes(N_n\parr M_n))\end{array}}{G''\parr P^{\perp}(\!|N_1,\ldots,N_n|\!)\parr P(\!|M_1,\ldots,M_n|\!)} \quad\text{or}\quad \mathsf{p}\!\downarrow\frac{\begin{array}{c}\varnothing\\ \mathcal{D}'\,\|\,\mathsf{GX}\\ G''\parr(J_1\otimes(J_2\parr L_2)\otimes\ldots\otimes(J_k\parr L_k))\end{array}}{G''\parr R^{\perp}(\!|J_1,\ldots,J_k|\!)\parr \dfrac{R(\!|\varnothing,L_2,\ldots,L_k|\!)}{P(\!|M_1,\ldots,M_n|\!)}}$$

  In the first case we conclude by Lemma 34, while the second requires Lemma 24 and inductive hypothesis to conclude.

- the last rule is a $\mathsf{q}\!\downarrow$ or $\mathsf{qm}$, that is, $\mathcal{D}$ is of the following shape for a prime web $Q$ with $\overset{Q}{\curvearrowright}\neq\varnothing$

$$\mathsf{q}\!\downarrow\frac{\begin{array}{c}\varnothing\\ \mathcal{D}'\,\|\,\mathsf{GX}\\ G''\parr Q^{*}(\!|J_1\parr L_1,\ldots,J_k\parr L_k|\!)\end{array}}{G''\parr Q^{\perp}(\!|J_1,\ldots,J_k|\!)\parr \dfrac{Q(\!|L_1,\ldots,L_k|\!)}{P(\!|M_1,\ldots,M_n|\!)}} \quad\text{or}\quad \mathsf{qm}\frac{\begin{array}{c}\varnothing\\ \mathcal{D}'\,\|\,\mathsf{GX}\\ G''\parr Q(\!|J_1\parr L_1,\ldots,J_n\parr L_k|\!)\end{array}}{G''\parr Q(\!|J_1,\ldots,J_k|\!)\parr \dfrac{Q(\!|L_1,\ldots,L_k|\!)}{P(\!|M_1,\ldots,M_n|\!)}}$$

  where $J_i\parr L_i\neq\varnothing$ for all $i\in\{1,\ldots,k\}$ and $Q^*$ denotes $Q^{\perp}$ if $\overset{Q}{\frown}=\varnothing$ and $Q$ otherwise. In this case, we conclude by induction hypothesis. Note that for the $\mathsf{q}\!\downarrow$ we do not have to take into consideration Case (C) since $\overset{Q^*}{\frown}\neq\varnothing$.

- We have the the special cases due to associativity of $\otimes$ and $\triangleleft$ concerning rules $\mathsf{s}_{\parr}$ and $\mathsf{q}\!\downarrow$.

$$\mathsf{s}_{\parr}\frac{G''\parr(M_1\otimes(G'\parr(M_2\otimes M_3)))}{G''\parr G'\parr\dfrac{M_1\otimes(M_2\otimes M_3)}{(M_1\otimes M_2)\otimes M_3}} \quad \mathsf{q}\!\downarrow\frac{G''\parr((N_1\parr M_1)\triangleleft(N_2\parr(M_2\triangleleft M_3)))}{G''\parr(N_1\triangleleft N_2)\parr\dfrac{M_1\triangleleft(M_2\triangleleft M_3)}{(M_1\triangleleft M_2)\triangleleft M_3}} \quad \mathsf{q}\!\downarrow\frac{G''\parr((N_1\parr(M_1\triangleleft M_2))\triangleleft(N_2\parr M_3))}{G''\parr(N_1\triangleleft N_2)\parr\dfrac{(M_1\triangleleft M_2)\triangleleft M_3}{M_1\triangleleft(M_2\triangleleft M_3)}}$$

  Note that a case similar to the first one above can be produced using $\mathsf{p}\!\downarrow$ instead of $\mathsf{s}_{\parr}$ and similar cases to the ones above on the right can be produced using $\mathsf{qm}$ instead of $\mathsf{q}\!\downarrow$, and, whenever one between $N_1$ or $N_2$ is empty, using $\mathsf{s}_{\parr}$ instead of $\mathsf{q}\!\downarrow$.

- The last rule is a sl, that is, $\mathsf{GX} = \mathsf{GV}^{\mathsf{sl}}$, $\overset{P}{\frown} = \varnothing$ and

$$\mathsf{sl}\,\frac{\begin{array}{c}\varnothing\\ \mathcal{D}' \,\|\, \mathsf{GX}\\ G \parr (P(\!|M_1,\ldots,M_k,\varnothing,\ldots,\varnothing|\!) \lhd P(\!|\varnothing,\ldots,\varnothing,M_{k+1},\ldots,M_n|\!))\end{array}}{G \parr P(\!|M_1,\ldots,M_n|\!)}$$

In this case, we can apply Lemma 34 and conclude by inductive hypothesis. ◀

**Sketch of Proof of Splitting Lemma for Complete Webs (Lemma 34).** By induction on $|V_{\mathbb{C}}|$ using Lemma 33. We know that all connectors and subconnectors of a complete graph are complete graphs, therefore webs. Therefore we can assume $\mathbb{C} = \mathbb{C}'(\!|\mathbb{C}_1,\ldots,\mathbb{C}_m|\!)$ for some $m$ and complete webs $\mathbb{C}', \mathbb{C}_1,\ldots,\mathbb{C}_m$ where $\mathbb{C}'$ is also prime. We can apply Lemma 33. If Case (A) of Lemma 33 applies, then we conclude by induction hypothesis. Otherwise Case (B) applies and we can assume without of loss of generality that we have $K_X$ and $K_Y$ such that $\vdash_{\mathsf{GX}} K_X \parr \mathbb{C}_1(\!|M_1,\ldots,M_l|\!)$ and $\vdash_{\mathsf{GX}} K_Y \parr \mathbb{C}''(\!|M_{l+1},\ldots,M_n|\!)$ for some $1 \leq l \leq n$, where $\mathbb{C}'' = \mathbb{C}'(\!|\varnothing,\mathbb{C}_2,\ldots,\mathbb{C}_m|\!)$. Since $\mathbb{C}''$ is a complete web, we can conclude by induction hypothesis. ◀

**Proof of Context-Reduction (Lemma 32).** If $A = \varnothing$ we conclude by letting $\mathcal{C}[\Box] = \varnothing$ and $K = \varnothing$. Otherwise, we can assume w.l.o.g. that $\mathcal{C}[A] = G \parr \mathcal{C}'[A]$ for a web $\mathcal{C}'[A]$ which is neither a par nor empty. If $\mathcal{C}'[\varnothing] = \varnothing$, then we can let $K = G$ and the derivation $\mathcal{D}_X$ is trivial. The derivation $\mathcal{D}_A$ is provided by assumption since $K \parr A = G \parr A = \mathcal{C}[A]$. If $\mathcal{C}'[\varnothing] = G'$ is non-empty, we proceed by induction on $\|\mathcal{C}'[A]\|$. Then w.l.o.g. we can assume that $\mathcal{C}[A] = G \parr P(\!|M_1[A], M_2,\ldots,M_n|\!)$ for a prime web $P \neq \parr$ and we can apply Lemma 33. We have the following three cases:

**(A)** There are webs $K_1,\ldots,K_n$ such that

$$\begin{array}{c}P^{\perp}(\!|K_1,\ldots,K_n|\!)\\ \mathcal{D}_G \,\|\, \mathsf{GX}\\ G\end{array} \;,\quad \begin{array}{c}\varnothing\\ \mathcal{D}_1 \,\|\, \mathsf{GX}\\ K_1 \parr M_1[A]\end{array} \quad\text{and}\quad \begin{array}{c}\varnothing\\ \mathcal{D}_i \,\|\, \mathsf{GX}\\ K_i \parr M_i\end{array}$$

for all $2 \leq i \leq n$.

**(B)** One of the two following sub-cases holds:

**(I)** there are webs $K_X$ and $K_Y$ such that

$$\begin{array}{c}K_X \parr K_Y\\ \mathcal{D}_G \,\|\, \mathsf{GX}\\ G\end{array} \;,\quad \begin{array}{c}\varnothing\\ \mathcal{D}_X \,\|\, \mathsf{GX}\\ K_X \parr M_1[A]\end{array} \quad\text{and}\quad \begin{array}{c}\varnothing\\ \mathcal{D}_Y \,\|\, \mathsf{GX}\\ K_Y \parr P(\!|\varnothing, M_2,\ldots,M_n|\!)\end{array}$$

**(II)** there are webs $K_X$ and $K_Y$ such that, w.l.o.g.,

$$\begin{array}{c}\varnothing\\ \mathcal{D}_X \,\|\, \mathsf{GX}\\ K_X \parr M_n\end{array} \;,\quad \begin{array}{c}\varnothing\\ \mathcal{D}_Y \,\|\, \mathsf{GX}\\ K_Y \parr P(\!|M_1[A], M_2,\ldots,M_{n-1},\varnothing|\!)\end{array} \quad\text{and}\quad \begin{array}{c}K_X \parr K_Y\\ \mathcal{D}_G \,\|\, \mathsf{GX}\\ G\end{array}$$

**(C)** $\mathsf{GX} = \mathsf{GV}^{\mathsf{sl}}$ and one of the two following sub-cases holds:

**(I)** there are webs $K_X$ and $K_Y$ such that

$$\begin{array}{c} K_X \triangleleft K_Y \\ \mathcal{D}_G \,\|\, \mathsf{GX} \\ G \end{array} \;,\; \begin{array}{c} \varnothing \\ \mathcal{D}_X \,\|\, \mathsf{GX} \\ K_X \parr P(\!|M_1, \ldots, M_{j-1}, M_j[A], M_{j+1}, \ldots, M_k, \varnothing, \ldots, \varnothing|\!) \end{array} \text{ and } \begin{array}{c} \varnothing \\ \mathcal{D}_Y \,\|\, \mathsf{GX} \\ K_Y \parr P(\!|\varnothing, \ldots, \varnothing, M_k, \ldots, M_n|\!) \end{array}$$

**(II)** The other case is similar but assuming $j > k$.

In all cases we can conclude by applying the induction hypothesis. ◀

**Sketch of Proof of Splitting Lemma for Atomic Webs (Lemma 31).** As in the proof of Lemma 33, we proceed by case analysis on the last rule in a derivation $\mathcal{D}$ of $G \parr a$.

- If rule $\mathsf{r}$ acts inside $G$ or the last rule in $\mathcal{D}$ is an $\mathsf{ai}{\downarrow}$, then the derivation $\mathcal{D}$ is of shape

$$\begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ \mathsf{r}\dfrac{G'}{G} \parr a \end{array} \quad \text{or} \quad \begin{array}{c} \varnothing \\ \mathcal{D}'_G \,\|\, \mathsf{GX} \\ G' \parr \mathsf{ai}{\downarrow}\dfrac{\varnothing}{a^\perp \parr a} \end{array}$$

  for some $\mathcal{D}'$. In the first case, we conclude by applying the induction hypothesis on $G' \parr a$ and in the second case, we let $\mathcal{D}_G = \mathcal{D}'_G \parr a^\perp$.

- If the last rule in $\mathcal{D}$ is a $\mathsf{s}_\parr$, then $\mathcal{D}$ is of one of the following shapes

$$\mathsf{s}_\parr \dfrac{\begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ P(\!|M_1, \ldots, M_{i-1}, a, M_{i+1}, \ldots, M_n|\!) \end{array}}{P(\!|M_1, \ldots, M_{i-1}, \varnothing, M_{i+1}, \ldots, M_n|\!) \parr a} \quad \text{or} \quad \mathsf{s}_\parr \dfrac{\begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ P(\!|M_1, \ldots, M_{i-1}, a \parr M_i, M_{i+1}, \ldots, M_n|\!) \end{array}}{P(\!|M_1, \ldots, M_n|\!) \parr a}$$

  In both cases we conclude by applying context reduction.

- If the last rule is a $\mathsf{p}{\downarrow}$, then w.l.o.g. $\mathcal{D}$ is of the shape

$$\begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ G'' \parr \mathsf{p}{\downarrow}\dfrac{(M_1 \parr a) \otimes M_2 \otimes \cdots \otimes M_n}{P(\!|M_1, \ldots, M_n|\!) \parr P^\perp(\!|a, \varnothing, \ldots, \varnothing|\!)} \end{array}$$

  We conclude by applying Lemma 34 and the induction hypothesis.

- If the last rule is a $\mathsf{q}{\downarrow}$, then w.l.o.g. $\mathcal{D}$ is of the shape

$$\mathsf{q}{\downarrow} \dfrac{\begin{array}{c} \varnothing \\ \mathcal{D}' \,\|\, \mathsf{GX} \\ G'' \parr Q^\perp(\!|M_1 \parr a, M_2, \ldots, M_n|\!) \end{array}}{Q^\perp(\!|M_1, \ldots, M_n|\!) \parr Q(\!|a, \varnothing, \ldots, \varnothing|\!)}$$

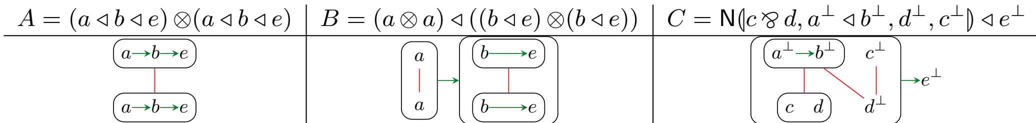
  for a complete prime web $Q^\perp$. We apply Lemma 33 and proceed as above.

- If the last rule is a $\mathsf{s}_\otimes$ or a $\mathsf{sl}$ then we conclude as in the first case. ◀

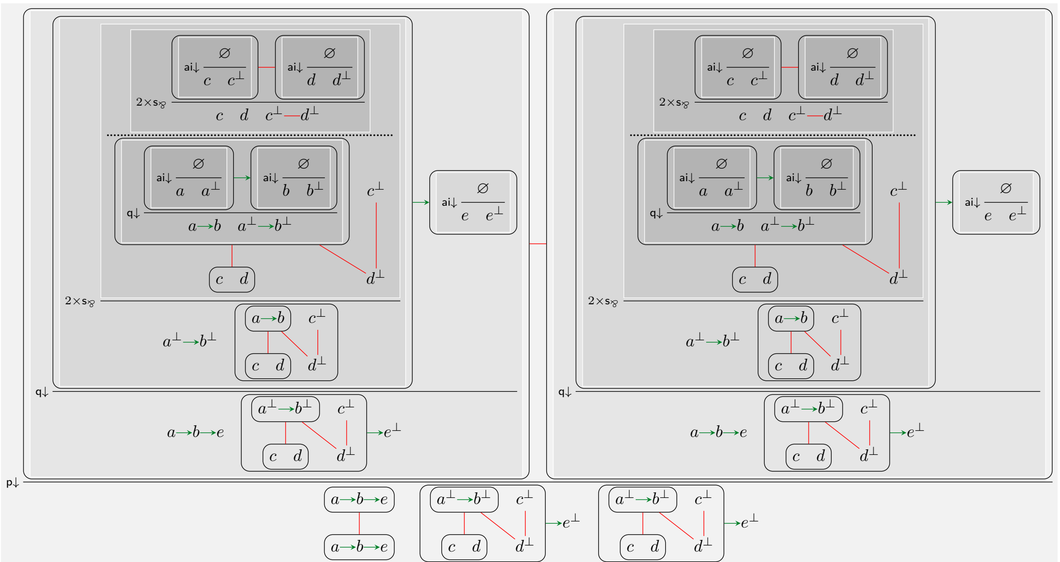
## B The Need for $\mathsf{s}_\otimes$ in GV

▶ **Example 44** ($\mathsf{s}_\otimes$ is not admissible.). The rule $\mathsf{q}_{\mathsf{BV}}\uparrow$ (which overlaps with $\mathsf{s}_\otimes$, as shown in Lemma 21) is admissible in BV [19]. Similarly, the rule $\mathsf{s}_\otimes$ is admissible in GS [1]. However, in [2] it was shown that $\mathsf{s}_\otimes$ is required to prove that GS is a conservative extension of the multiplicative linear logic (MLL). Furthermore, we are able to provide an example showing that $\mathsf{s}_\otimes$ is not admissible in any of our systems conservatively extending GS and BV. Consider the following webs.

$A = (a \triangleleft b \triangleleft e) \otimes (a \triangleleft b \triangleleft e)$ | $B = (a \otimes a) \triangleleft ((b \triangleleft e) \otimes (b \triangleleft e))$ | $C = \mathsf{N}(\!| c \parr d, a^\perp \triangleleft b^\perp, d^\perp, c^\perp |\!) \triangleleft e^\perp$



We have $\vdash_{\mathsf{BV}} A \multimap B$ and $\vdash_{\mathsf{GV}} A \parr C \parr C$ as shown below.



By cut elimination for GV, we get $\vdash_{\mathsf{GV}} B \parr C \parr C$. However, $\vdash_{\mathsf{GV}\setminus\{\mathsf{s}_\otimes\}} B \parr C \parr C$ does not hold.

# A Stratified Approach to Löb Induction


**Daniel Gratzer**
Aarhus University, Denmark

**Lars Birkedal**
Aarhus University, Denmark



## Abstract

Guarded type theory extends type theory with a handful of modalities and constants to encode productive recursion. While these theories have seen widespread use, the metatheory of guarded type theories, particularly guarded *dependent* type theories remains underdeveloped. We show that integrating Löb induction is the key obstruction to unifying guarded recursion and dependence in a well-behaved type theory and prove a no-go theorem sharply bounding such type theories.

Based on these results, we introduce GuTT: a stratified guarded type theory. GuTT is properly two type theories, sGuTT and dGuTT. The former contains only propositional rules governing Löb induction but enjoys decidable type-checking while the latter extends the former with definitional equalities. Accordingly, dGuTT does not have decidable type-checking. We prove, however, a novel *guarded canonicity* theorem for dGuTT, showing that programs in dGuTT can be run. These two type theories work in concert, with users writing programs in sGuTT and running them in dGuTT.



**2012 ACM Subject Classification** Theory of computation → Type theory; Theory of computation → Denotational semantics; Theory of computation → Modal and temporal logics

**Keywords and phrases** Dependent type theory, guarded recursion, modal type theory, canonicity, categorical gluing

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.23

**Funding** Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.


## 1 Introduction

It is well-known that a fixed-point combinator $\mathtt{fix} : (A \to A) \to A$ wreaks havoc within type theory by rendering the theory unsound and making type-checking undecidable. Instead of a unified concept of recursion therefore, type theory is extended with sound induction and coinduction principles with implementations using syntactic checks to ensure that recursively-defined functions can be elaborated into these tamer principles.

Unfortunately, these syntactic checks are brittle, often forcing an artificial restructuring of a program. In order to obtain a type-theoretic account of recursion, Nakano [20] introduced *guarded recursion*. Instead of ensuring *a posteriori* that recursive calls occur after progress has been made, guarded recursion changes the type of recursive calls to prevent non-productive use.

Concretely, guarded type theory extends type theory with a modality – a unary type constructor – $\blacktriangleright A$ (pronounced "later $A$"). This modality is cartesian and comes with a point $\mathsf{next} : A \to \blacktriangleright A$. It does not, however, have an operator $\blacktriangleright A \to A$; once data enters the $\blacktriangleright$ modality, it cannot be extracted again. The fixed-points are then restored by Löb induction:

$$\mathsf{lob} : (\blacktriangleright A \to A) \to A \qquad\qquad \mathsf{lob}(f) = f(\mathsf{next}(\mathsf{lob}(f)))$$

While Löb induction was introduced in the context of provability, the $\blacktriangleright$ modality has a temporal intuition: $\blacktriangleright A$ contains elements of $A$ available one step in the future. As it stands, however, lob prevents non-productive usage a little too well and there are no types


© Daniel Gratzer and Lars Birkedal;
licensed under Creative Commons License CC-BY 4.0


7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 23; pp. 23:1–23:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with non-constant maps $\blacktriangleright A \to A$ which renders $\mathsf{lob}$ induction useless. In order to rectify this, guarded type theories introduce variants of common types instrumented with $\blacktriangleright$s. The perennial example of this is the guarded stream type:

$$\texttt{GStr}_A = A \times \blacktriangleright \texttt{GStr}_A \tag{1}$$

After isolating the tail of the stream under $\blacktriangleright$, stream processors are defined using Löb:

$$\texttt{map}(f) = \mathsf{lob}(\lambda r.\ \lambda s.\ (f(\mathsf{pr}_1(s)), r \circledast \mathsf{pr}_2(s)))$$

Here $(\circledast) : \blacktriangleright(A \to B) \to \blacktriangleright A \to \blacktriangleright B$ is an operator which witnesses the fact that $\blacktriangleright$ is cartesian. A non-productive stream processor such as `filter` cannot be defined: any attempt to define `filter` leaves us with $\blacktriangleright \texttt{Str}_A$ in a place where we require $\texttt{Str}_A$.

On top of this substrate, many variants of guarded type theory have been proposed [1, 2, 4, 5, 7, 12, 19, 22]. These extend the theory with guarded recursive types, more intricate combinations of modalities, or more flexible notions of time. In particular, extending the theory with an idempotent comonad $\square$ such that $\square \blacktriangleright A \simeq \square A$ allows guarded type theories to encode coinductive types through guarded recursive types [7].

Unfortunately, attempts to extend guarded type theory to dependent types have been met with mixed success. On one hand they escape the inconsistencies of an unguarded fixed-point combinator, but on the other this alone does not make type-checking decidable.

## 1.1 Modal dependent type theory

In prior work [4, 5, 19], the precise cause of this undecidability issue was muddied. Type theorists had only just begun to understand modalities in dependent type theory, and the inclusion of any modality was apt to break properties such as decidability of type-checking. Independent of guarded type theories, modal type theory have been the subject of intense focus and recently Gratzer et al. [10] have proposed $\mathsf{MTT}$, a calculus which enjoys normalization, canonicity, and decidable type-checking [9] and which can support $\blacktriangleright$ and $\square$ (but not $\mathsf{lob}$).

$\mathsf{MTT}$ is a *Fitch-style* modal type theory [3] parameterized by a 2-category of modes, modalities, and coercions between them. In this theory, modalities (written $\langle \mu \mid A \rangle$) are equipped with an "adjoint" action on contexts $\Gamma.\{\mu\}$. The introduction rules for modalities are given by tranposition along the "adjunction" $-.\{\mu\} \dashv \langle \mu \mid - \rangle$:

$$\frac{\Gamma.\{\mu\} \vdash M : A}{\Gamma \vdash \mathsf{mod}_\mu(M) : \langle \mu \mid A \rangle}$$

$\mathsf{MTT}$ can be instantiated with two modalities $\ell$ and $e$ along with equations forcing $\ell$ to model $\blacktriangleright$ while $\langle e \mid - \rangle$ becomes its left adjoint. The left adjoint to $\blacktriangleright$, written $\blacktriangleleft$ by Birkedal et al. [4], can be used to *encode* the more familiar $\square$ modality as the limit of $\blacktriangleleft$, $\blacktriangleleft^2$, etc., as in the standard model of guarded recursion in $\mathbf{PSh}(\omega)$ (presheaves over $\omega$).

Even with these advances, however, a comprehensive guarded dependent type theory has remained out of reach. Adding Löb induction directly to $\mathsf{MTT}$ disrupts canonicity and normalization, and other proposed guarded type theories either fail to support crucial reasoning principles [2] or lack a proof of canonicity or normalization [17].

In fact, Löb induction has proven to be a far more serious obstacle to a well-behaved guarded type theory than the modal apparatus. We will prove a "no-go" result that shows that including a definitional equality allowing Löb induction to unfold nearly always results in undecidable type-checking. This effectively rules out a theory which naïvely includes $\mathsf{lob}$.

## 1.2 Stratifying guarded type theory

Given the complications of including Löb induction into guarded type theory, we propose a novel approach to the problem with a type theory GuTT stratified into two layers: static and dynamic. Both of the two layers of the theory exhibit one of the two key properties of a type theory (normalization and canonicity), but neither satisfies both.

The static layer of our system does not include any definitional equalities for Löb induction and only supplies the user with propositional equalities. This layer includes not only ► and lob, but an ◄ modality which enables the user to encode coinductive types. Traditionally, the inclusion of ◄ and ► satisfying the correct properties has been a significant source of difficulty for type theorists, but the aforementioned advances in modal type theory ensure that this static system enjoys decidable type-checking. Moreover, even without definitional equalities for Löb induction we have shown the static layer to be highly usable.

By including these axioms without the corresponding definitional equalities, however, the static layer does not validate canonicity; there are closed terms of type nat which are not convertible to numerals. Our "no-go" theorem implies that rectifying this problem would require a radical restructuring of GuTT. Accordingly, we introduce a second "dynamic" layer to our theory. The dynamic system is an extension of the static theory by definitional equalities making lob compute and forcing the propositional equality governing Löb induction to collapse to reflexivity. The dynamic theory does not enjoy decidable type-checking, but the additional equalities cause previously stuck terms to compute which we have proven via a novel "guarded" canonicity theorem.

In total, a user may write a term in the static theory and type-check it as expected and then extract it to the dynamic theory for computation. By a careful analysis of the models of these theories, we show that computation can be used as a valid reasoning principle when viewing the type theory as the internal language of a model such as the topos of trees.

## 1.3 Guarded canonicity

Typically, canonicity states that a closed term in type theory is convertible with a canonical form. In guarded type theory, this is complicated by two factors.

The first is the presence of modal types like $\blacktriangleright A$: what are the canonical forms of $\blacktriangleright A$? The answer depends on the precise formulation of modalities, but existing solutions from MTT [9] apply. Specifically, we prove a canonicity result not just for terms in context $\mathbf{1}$, but in contexts $\mathbf{1}.\{\mu\}$ for all $\mu$. This generalization makes it easy to state the canonical forms of $\langle \mu \mid A \rangle$ in context $\mathbf{1}.\{\nu\}$: they are terms $\mathsf{mod}_\mu(M)$ such that $M$ is canonical in $\mathbf{1}.\{\nu \circ \mu\}$.

More seriously, in the presence of Löb induction canonical elements of some types must be infinite. In order to tame this non-termination, we adopt an idea common in guarded recursion and allow only finite approximations of infinite canonical forms [12]. dGuTT is extended with a special context $\mathbf{0}$ under which all judgments collapse. By indexing $\mathbf{0}$ with a modality and ensuring $\mathbf{0}[\mu].\{\mu\} \cong \mathbf{0}$, we can prove canonicity only to a certain "depth", without counting steps or otherwise imposing any rewriting system on our type theory. We prove guarded canonicity through an adaptation of *multimodal synthetic Tait computability* [9, 26].

## 1.4 Contributions

We prove the first "no-go" theorem bounding the possibilities for including Löb induction in a type theory with decidable type-checking. As an answer to this result, we contribute GuTT, a guarded type theory stratified into static and dynamic layers. We show that the static layer has decidable type-checking and that the dynamic layer satisfies a novel form of canonicity.

In Section 2 we motivate and prove the no-go theorem for Löb induction. In Section 3 we introduce GuTT and prove that the static layer enjoys decidable type-checking. In Section 4 we state and prove the first guarded canonicity result for a guarded type theory. Finally, in Section 5 we show that even without certain definitional equalities the static layer is highly usable and formalize a model of idealized synchronous programming within sGuTT.

## 2 A no-go theorem for Löb induction

Consider the essential ingredients of a guarded type theory specified above: Martin-Löf type theory equipped with a pointed cartesian modality $(\blacktriangleright, \mathsf{next}, \circledast)$ and a constant $\mathsf{lob} : (\blacktriangleright A \to A) \to A$ satisfying the following definitional equality:

$$\mathsf{lob}(f) = f(\mathsf{next}(\mathsf{lob}(f)))$$

We show that under two minor additional assumptions, definitional equality – and therefore type-checking – is undecidable in this system. Our first assumption is the existence of a non-trivial guarded type. Specifically, a type $S$ equipped with a definitional isomorphism:

$$\mathtt{cons} : \mathsf{S} \cong \mathsf{nat} \times \blacktriangleright \mathsf{S} \tag{2}$$

In the presence of universes and a suitable *dependent* version of later $\hat{\blacktriangleright} : \blacktriangleright \mathsf{U} \to \mathsf{U}$, we can define $S = \mathsf{lob}(x.\ \mathsf{nat} \times \hat{\blacktriangleright} x)$. In order to make this counter-example as broad as possible, however, we have chosen to explicitly isolate $S$.

The second assumption is that $\mathsf{next}$ must be suitably injective with respect to definitional equality. In particular, $\mathbf{1} \vdash \mathsf{next}(M) = \mathsf{next}(N) : \blacktriangleright A$ must be equivalent to $\mathbf{1} \vdash M = N : A$. This second requirement is more onerous than the first, and it does not follow directly from the existence of a familiar structure like universes. It does hold, however, in the standard model of guarded recursion $\mathbf{PSh}(\omega)$ [4]. Additionally, in a guarded type theory extended with $\square$ or $\blacktriangleleft$, this bi-implication is derivable. In fact, while initially surprising, this second assumption is satisfied by all guarded type theories in the literature.

With these requirements, we show that deciding conversion would entail deciding the extensional equality of functions $\mathsf{nat} \to \mathsf{nat}$.

▶ **Lemma 1.** *There exists a function* $\mathtt{tabulate} : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{S}$ *such that the nth element of* $\mathtt{tabulate}(f)$ *is* $\mathsf{next}^n(f(\bar{n})) : \blacktriangleright^n \mathsf{nat}$.

**Proof.** We define $\mathtt{tabulate}$ by means of a helper function using Löb induction:

$$\mathsf{go} : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{S}$$
$$\mathsf{go}(f) = \mathsf{lob}(\lambda r, n.\ \mathtt{cons}(f(n), r \circledast \mathsf{next}(n+1)))$$
$$\mathtt{tabulate}(f) = \mathsf{go}(f, 0)$$
◀

▶ **Theorem 2.** *Given two closed functions* $f, g : \mathsf{nat} \to \mathsf{nat}$*, the streams* $\mathtt{tabulate}(f)$ *and* $\mathtt{tabulate}(g)$ *are convertible if and only if* $f(\bar{n}) = g(\bar{n})$ *for all* $n : \mathbb{N}$.

**Proof.** The definitional isomorphism $\mathtt{cons}$ together with the $\eta$ principle for dependent sums ensure that it is both necessary and sufficient to show that the two streams are pointwise equal, but Lemma 1 together with the injectivity of $\mathsf{next}$ on closed terms reduces this condition to the extensional equality of $f$ and $g$. ◀

$$\ell : m \longrightarrow m \qquad e : m \longrightarrow m$$

$$\mathsf{id} = e \circ \ell \qquad \mathsf{id} \leq \ell \circ e \qquad \mathsf{id} \leq \ell$$

**Figure 1** $\mathcal{M}$: a mode theory for guarded recursion.

▶ **Corollary 3.** *Conversion is undecidable in a guarded type theory satisfying the following requirements:*

- *there is a type $S$ equipped with a definitional isomorphism $S \cong \mathsf{nat} \times \blacktriangleright S$.*
- $\mathsf{next} : A \to \blacktriangleright A$ *is injective on closed terms,*
- *the* lob *operator unfolds.*

# 3 GuTT: a stratified guarded type theory

Corollary 3 places firm bounds on how much a guarded type theory can hope to achieve while maintaining decidable type-checking. In particular, it shows that including an unconditional unfolding rule for Löb induction makes this goal practically untenable, but without the ability to unfold lob there is no hope for any canonicity or computational adequacy result. While some guarded type theories have approached this problem by attempting to restrict the behavior of Löb induction to compute only in certain situations [2, 17], GuTT takes a more radical approach.

Instead of considering a single theory which is carefully crafted to enjoy both computational adequacy and decidable type-checking simultaneously, GuTT is split into a pair of type theories: sGuTT and dGuTT. The first enjoys decidable type-checking and the second satisfies (guarded) canonicity, but neither type theory has both properties.

Both type theories are extensions of MTT [10] with a mode theory $\mathcal{M}$ for guarded recursion detailed in Figure 1. For compatibility with MTT, this mode theory is presented as a 2-category, but contains only one object and is poset-enriched, so that all 2-cells are uniquely determined by their boundaries. In fact, $\mathcal{M}$ could equivalently be described as a monoidal partial order with two generating modalities $\ell$ and $e$. The inequalities governing these modalities force $\ell$ (respectively $e$) to behave like $\blacktriangleright$ (resp. $\blacktriangleleft$) on $\mathbf{PSh}(\omega)$.

## 3.1 Posetal MTT

Given that both sGuTT and dGuTT are based around this instantiation of MTT, we briefly review some of the key rules of MTT specialized to this mode theory. We refer the reader to Gratzer et al. [10] for a more comprehensive introduction to MTT. As a Fitch-style type theory, each modality in MTT behaves like an adjoint – in fact, a dependent right adjoint [3] – with the left adjoint acting on contexts. For a given modality $\mu$, we write the action of this left adjoint $\Gamma.\{\mu\}$. In fact, this action extends to a 2-functor from $\mathcal{M}^{\mathsf{coop}}$ to the category of contexts and substitutions,[1] complete with a functorial action $\gamma \mapsto \gamma.\{\mu\}$ on substitutions and definitional equalities $\Gamma.\{\nu \circ \mu\} = \Gamma.\{\nu\}.\{\mu\}$. Moreover, an inequality of modalities $\mu \leq \nu$ induces a natural transformation $-.\{\nu\} \longrightarrow -.\{\mu\}$.

[1] $\mathcal{M}^{\mathsf{coop}}$ is a 2-category with the same objects as $\mathcal{M}$, but with the directions of 1 and 2-cells reversed.

▶ Remark 4. While formally we follow [10] and work with MTT, sGuTT, and dGuTT as generalized algebraic theories – complete with explicit substitutions, de Bruijn indices, etc. – for exposition's sake with work with a more familiar informal syntax in examples.

As was mentioned in Section 1.1, the introduction rule for the modality $\langle \mu \mid - \rangle$ is then inspired by transposition:

$$\frac{\Gamma.\{\mu\} \vdash M : A}{\Gamma \vdash \mathsf{mod}_\mu(M) : \langle \mu \mid A \rangle}$$

Here we already have taken advantage of the particulars of $\mathcal{M}$ to simplify the rules of MTT; in general, a mode theory might contain multiple modes (multiple objects in the 2-category) which would require us to annotate all the judgments of the type theory by a mode. Given that we work with only one mode, however, we omit these annotations.

The elimination rule is more complex; adding the inverse direction of transposition would disrupt the substitution properties of the type theory. Instead, each entry in the context is annotated by a modality $\Gamma, x : (\mu \mid A)$. The rule for accessing variables is then tweaked to factor in both the annotation on a variable and the modalities following it in the context:

$$\frac{\mu \leq \mathsf{mods}(\Gamma_1)}{\Gamma_0, x : (\mu \mid A), \Gamma_1 \vdash x : A}$$

Here $\mathsf{mods}(\Gamma_1)$ is the composition of all modalities $\{\nu\}$ occurring in $\Gamma_1$. Unlike full MTT, variables are not annotated by 2-cells. As $\mathcal{M}$ is merely enriched over posets, at most one 2-cell can exist for any given pair of modalities. Accordingly, we remove these annotations on variables and replace them with the premise $\mu \leq \nu$ in the variable rule. In fact, given an inequality $\mu \leq \nu$ we will silently coerce from terms and types in context $\Gamma.\{\mu\}$ to terms and types in context $\Gamma.\{\nu\}$. No ambiguity can arise from this because $\mathcal{M}$ is poset-enriched.

Dependent products are in turn generalized to allow for abstraction over elements of $A$ annotated with a modality other than id:

$$\frac{\Gamma, x : (\mu \mid A) \vdash M : B}{\Gamma \vdash \lambda x.\ M : (\mu \mid x : A) \to B} \qquad \frac{\Gamma \vdash M : (\mu \mid x : A) \to B \qquad \Gamma.\{\mu\} \vdash N : A}{\Gamma \vdash M(N) : B[N/x]}$$

Finally, the addition of annotations for variables in the context introduces some redundancy in the system: what is the relationship between a variable of type $\langle \mu \mid A \rangle$ annotated with $\nu$ and a variable of type $A$ annotated with $\nu \circ \mu$. The role of the elimination rule for $\langle \mu \mid - \rangle$ is to address this mismatch and patch the difference between $\Gamma.(\nu \mid \langle \mu \mid A \rangle)$ and $\Gamma.(\nu \circ \mu \mid A)$:

$$\frac{\begin{array}{cc} \Gamma.\{\nu \circ \mu\} \vdash A & \Gamma, x : (\nu \mid \langle \mu \mid A \rangle) \vdash B \\ \Gamma.\{\nu\} \vdash M_0 : \langle \mu \mid A \rangle & \Gamma, x : (\nu \circ \mu \mid A) \vdash M_1 : B[\mathsf{mod}_\mu(x)/x] \end{array}}{\Gamma \vdash \mathsf{let}_\nu\ \mathsf{mod}_\mu(x) \leftarrow M_0\ \mathsf{in}\ M_1 : B[M_0/x]}$$

These primitives combine to ensure that e.g. modal types become a "pseudofunctor" so that $\mathsf{comp}[\mu;\nu] : \langle \mu \circ \nu \mid A \rangle \simeq \langle \mu \mid \langle \nu \mid A \rangle \rangle$ and an inequality $\mu \leq \nu$ induces a coercion $\mathsf{coe}[\mu \leq \nu] : \langle \mu \mid A \rangle \to \langle \nu \mid A \rangle$. We introduce some shorthand for this instantiation of MTT:

$$\blacktriangleright A \triangleq \langle \ell \mid A \rangle \qquad \blacktriangleleft A \triangleq \langle e \mid A \rangle$$

The pseudofunctorial structure of modalities quickly yields the standard operations of a guarded type theory e.g. $\mathsf{now} : \blacktriangleleft \blacktriangleright A \simeq A$ and $\mathsf{next} : A \to \blacktriangleright A$. For instance,

$$\mathsf{next}_A = \lambda x.\ \mathsf{mod}_\ell(x)$$

## 3.2 Crisp identity induction principles

In MTT, the elimination principles for types like $\mathsf{Id}_A(M, N)$ or $\mathsf{bool}$ are *mode-local* and therefore can only be applied to a variable if it is annotated with $\mathsf{id}$. This restriction is vital for soundness; the intended model of GuTT does not validate $\langle \ell \mid \mathsf{bool} \rangle \simeq \mathsf{bool}$ and allowing $\mathsf{if}\ x\ \mathsf{then}\ M_0\ \mathsf{else}\ M_1$ when $x$ is under an $\ell$ annotation is equivalent to such an identification.

The intended model does, however, validate such an equivalence for the identity type. Intuitively, the typical models of GuTT are extensional and accordingly $\mathsf{Id}_A(M, N)$ is realized by an equalizer (a limit) which is therefore preserved by all modalities (right adjoints):

$$\mathsf{Id}_{\langle \mu \mid A \rangle}(\mathsf{mod}_\mu(M), \mathsf{mod}_\mu(N)) \simeq \langle \mu \mid \mathsf{Id}_A(M, N) \rangle \tag{3}$$

As it turns out, this equivalence is crucial in practice. While we will see specific examples in Section 5, informally any proof of equality between programs defined via Löb induction requires Equation (3). In order to prove e.g. that two stream processors are equal, we must show that they produce streams which have identical heads and tails. Löb induction gives a proof under a later that the two tails are equal, but without Equation (3) we cannot parlay this into an equality between $\mathsf{next}$ applied to the two tails.

In prior guarded variants of MTT, this was circumvented by adopting equality reflection, but GuTT takes a more refined approach and adds a stronger variant of the elimination principle for identity types inspired by Shulman's crisp induction principles [24]:

$$\frac{\begin{array}{c}\Gamma, a_0 : (\mu \mid A), a_1 : (\mu \mid A), p : (\mu \mid \mathsf{Id}_A(a_0, a_1)) \vdash B \\ \Gamma, x : (\mu \mid A) \vdash M : B[x, x, \mathsf{refl}(a)/a_0, a_1, p] \\ \Gamma.\{\mu\} \vdash N_0, N_1 : A \qquad \Gamma.\{\mu\} \vdash P : \mathsf{Id}_A(N_0, N_1)\end{array}}{\Gamma \vdash \mathsf{J}^\mu(B, a.M, P) : B[N_0, N_1, P/a_0, a_1, p]} \qquad \mathsf{J}^\mu(B, x.M, \mathsf{refl}(N)) = M[N/x]$$

Gratzer [9] has shown that extending MTT with these rules preserves both canonicity and normalization. This, together with the fact that all intended models of GuTT validate the equivalent Equation (3), ensures that adding these rules to guarded MTT has no downside.

## 3.3 sGuTT: The static fragment of GuTT

The "static" fragment of the guarded calculus adds Löb induction to guarded MTT. It enjoys decidable type-checking but escapes Corollary 3 by not including any definitional equalities for the $\mathsf{lob}$ operator. Concretely, sGuTT is the instantiation of MTT described above supplemented with the following pair of axioms:

$$\frac{\Gamma, x : (\ell \mid A) \vdash M : A}{\Gamma \vdash \mathsf{lob}(x.M) : A} \qquad \frac{\Gamma, x : (\ell \mid A) \vdash M : A}{\Gamma \vdash \mathsf{unfold}(M) : \mathsf{Id}_A(\mathsf{lob}(x.M), M[\mathsf{lob}(x.M)/x])}$$

A term in sGuTT is then precisely a term in MTT in a context extended by these two constants, so we may immediately deduce the following results from the results of Gratzer [9]:

▶ **Theorem 5.** *Conversion and type-checking in sGuTT are both decidable sGuTT.*

**Proof.** Corollaries 6.5 and 6.6 of Grater [9] show that conversion and type-checking are decidable if the equality of 1- and 2-cells $\mathcal{M}$ enjoy decidable conversion. This problem is easily seen to be equivalent to the decidability of the $\leq$ relation presented in Figure 1. We first observe that every 1-cell can be uniquely presented as $l^n \circ e^m$ for some pair $(n, m)$. Next, because $l^{n_0} \circ e^{m_0} \leq l^{n_1} \circ e^{m_1}$ if and only if $n_0 \leq n_1$ and $m_1 - n_1 \leq m_0 - n_0$, the equality of 1-cells and existence of 2-cells are both decidable as required. ◀

### 3.4 dGuTT: the dynamic fragment of GuTT

The dynamic half of GuTT is an extension of sGuTT, supplementing the static theory with a pair of definitional equalities:

$$\frac{\Gamma, x : (\ell \mid A) \vdash M : A}{\begin{array}{c}\Gamma \vdash \mathsf{lob}(x.M) = M[\mathsf{lob}(x.M)/x] : A \\ \Gamma \vdash \mathsf{unfold}(x.M) = \mathsf{refl}(\mathsf{lob}(x.M)) : \mathsf{Id}_A(\mathsf{lob}(M), M[\mathsf{lob}(x.M)/x])\end{array}}$$

Notice that Corollary 3 applies to dGuTT and so type-checking dGuTT is undecidable.

We also equip dGuTT with a new form of context $\mathbf{0}[\mu]$, which will prove crucial to understanding its computational behavior. The context $\mathbf{0} = \mathbf{0}[\mathsf{id}]$ acts like an initial object in the category of contexts so that all judgments $\mathbf{0} \vdash \mathcal{J}$ hold, and the more general $\mathbf{0}[\mu]$ intuitively represents the application of the modality $\mu$ to this initial object. The rules for these contexts are presented in Figure 2, but we postpone further discussion of $\mathbf{0}$ until Section 4 where its role can be fully motivated.

Henceforth, it will become important to distinguish between which judgments hold in sGuTT and which hold in dGuTT. We write $\Gamma \vdash_{\mathbf{s}} M : A$ if $M : A$ is derivable with only the rules of sGuTT and $\Gamma \vdash_{\mathbf{d}} M : A$ for the corresponding notion for dGuTT. As dGuTT is a strict extension of sGuTT, the following is immediate:

▶ **Theorem 6.** *If a judgment is derivable in sGuTT, the same judgment holds in dGuTT.*

### 3.5 The semantics of GuTT

A priori, of course, Theorem 6 could hold even if dGuTT included all manner of extensions to sGuTT. The fact that it only extends sGuTT with two additional equations means that all models of dGuTT are models of sGuTT and all "natural" semantic models of sGuTT are also model of dGuTT. Concretely, the category of models of sGuTT is precisely the category of models of MTT [10] with mode theory $\mathcal{M}$ together with constants for lob and unfold. Unfolding these definitions, a model of sGuTT is a category with families [8] equipped with functors interpreting $-.\{\mu\}$ and structure for interpreting modal types.

A model of dGuTT is a model of sGuTT satisfying a pair of additional equalities and with structure interpreting $\mathbf{0}[\mu]$. In practice these additional requirements are easily satisfied:

▶ **Theorem 7.** *A model of sGuTT satisfying the following extends to a model of dGuTT:*
1. *the category of contexts and substitutions has an initial object*
2. *the interpretation of* Id *validates equality reflection*
3. *the model is democratic; every context can be realized as a type*
4. *the modalities are realized by dependent right adjoints [3, Definition 2]*

In particular, the standard models of guarded recursion in sheaves over complete Heyting algebras with a well-founded basis such as $\mathbf{PSh}(\omega)$ are models of both sGuTT and dGuTT [4]. See Appendix A for a full definition of models of sGuTT and dGuTT.

## 4 Guarded canonicity

We have shown that sGuTT enjoys decidable type-checking. In this section we establish that the twin theory dGuTT is computationally effective, i.e. that programs in this theory can run. Typically, a canonicity theorem for type theory states that any closed boolean is convertible with either true or false. For dGuTT, this result would be unsatisfactory because it would

$$\frac{}{\mathbf{0}\ \mathsf{cx}} \qquad \frac{}{\mathbf{0}[\mu]\ \mathsf{cx}} \qquad \frac{}{\mathbf{0} \vdash \lightning : A} \qquad \frac{}{\mathbf{0} \vdash \lightning} \qquad \frac{}{\mathbf{0}.\{\mu\} \vdash \lightning : \Gamma}$$

$$\frac{\Gamma \vdash \gamma : \mathbf{0}}{\Gamma \vdash M = N : A \qquad \Gamma \vdash A = B \qquad \Gamma \vdash \delta_0 = \delta_1 : \Delta} \qquad \frac{\Gamma \vdash \gamma : \mathbf{0}[\mu]}{\Gamma.\{\mu\} \vdash \gamma^{\leftarrow} : \mathbf{0}} \qquad \frac{\Gamma.\{\mu\} \vdash \gamma : \mathbf{0}}{\Gamma \vdash \gamma^{\rightarrow} : \mathbf{0}[\mu]}$$

$$\gamma^{\leftarrow\rightarrow} = \gamma \qquad \gamma^{\rightarrow\leftarrow} = \gamma \qquad \gamma^{\leftarrow} \circ \delta = (\gamma \circ \delta.\{\mu\})^{\leftarrow}$$

**Figure 2** The vacuous context.

not give information about terms of $\blacktriangleright$ bool and therefore could not shed light on executing guarded programs. We therefore characterize canonical forms in a class of contexts $\mathcal{C}$ other than the empty context. By ensuring that $\mathcal{C}$ is closed under $-.\{\mu\}$, canonicity applies to programs executing under a $\blacktriangleright$, but this raises a further complication in the presence of Löb induction: programs like `tabulate`(id) (Lemma 1) with infinite canonical forms. Rather than allowing the canonical forms themselves to be infinite, we use a novel type-directed form of "fuel" in our statement of canonicity.

Prior to formulating this more refined notion, recall from Section 3.4 that dGuTT has a distinguished context $\mathbf{0}$ which is initial in the category of contexts and substitutions (Figure 2). For each modality $\mu$, there exists an additional context $\mathbf{0}[\mu]$ – intuitively this is the context $\mathbf{0}$ under the modality $\mu$ – whose behavior is fully captured by a natural isomorphism of hom-sets in the category of contexts: $\hom(\Gamma, \mathbf{0}[\mu]) \cong \hom(\Gamma.\{\mu\}, \mathbf{0})$. In order to force this isomorphism, we add new substitution formers to our calculus: $\gamma^{\leftarrow}$ and $\gamma^{\rightarrow}$.

With this additional structure, we then characterize the canonical forms in programs in context $\mathbf{0}[\mu].\{\nu\}$ so that $(\mu, \nu)$ serves as an abstracted form of fuel:

▶ **Theorem 20** (Guarded canonicity). *Given a term $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} M : A$, the following holds*

- *If $A = \mathsf{nat}$ then there exist a numeral $k$ such that $M = \bar{k}$*
- *If $A = \langle \xi \mid B \rangle$ then $M = \mathsf{mod}_{\xi}(N)$ for some $\mathbf{0}[\mu].\{\nu \circ \xi\} \vdash_{\mathbf{d}} N : B$*
- *If $A = \mathsf{Id}_B(b_0, b_1)$ then $M = \mathsf{refl}(b_0)$ and $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} b_0 = b_1 :$.*

▶ Remark 8. Theorem 20 characterizes the canonical forms of all types in dGuTT – not merely nat, $\langle - \mid - \rangle$, and $\mathsf{Id}_{-}(-,-)$. This characterization is trivial, however, for types with an $\eta$ law. For instance, every element of $(\mu \mid A) \to B$ is of the form $\lambda(M)$ by $\eta$-expansion. We have further elided the characterization of canonical forms of the universe for reasons of space.

We pause for a moment to consider the consequences of Theorem 20 for various types and pairs $(\mu, \nu)$. First, if $\mu = \nu$ then there is a canonical substitution $\mathbf{0}[\mu].\{\nu\} \vdash \mathsf{id}^{\leftarrow} : \mathbf{0}$ and canonicity trivializes – all types are inhabited and all terms are equal in such a context.

Suppose instead, therefore, we have a term $\mathbf{0}[\ell \circ \ell].\{\mathsf{id}\} \vdash_{\mathbf{d}} M : \langle \ell \mid \langle \ell \mid A \rangle \rangle$. Guarded canonicity then ensures that $M = \mathsf{mod}_{\ell}(M_0)$ for some $\mathbf{0}[\ell \circ \ell].\{\ell\} \vdash_{\mathbf{d}} M_0 : \langle \ell \mid A \rangle$. The theorem then applies to $M_0$, yielding $M_0 = \mathsf{mod}_{\ell}(M_1)$ for some $\mathbf{0}[\ell \circ \ell].\{\ell \circ \ell\} \vdash_{\mathbf{d}} M_1 : A$ but the process stops here; the context is now isomorphic to $\mathbf{0}$ and so $M_1 = \lightning$.

Informally, in a context $\mathbf{0}[\mu].\{\nu\}$ the modality $\mu$ represents the initial fuel given to evaluate a program, while $\nu$ signifies the fuel expended thus far in the process. The analogy is imperfect – the presence of the earlier modality allows for "negative" fuel to be spent – but it does make it clear how one might use Theorem 20: to evaluate a closed program $M : \blacktriangleright^n \mathsf{nat}$,

one begin by weakening the program into the context $\mathbf{0}[\ell^{n+1}]$ and then repeatedly apply guarded canonicity to deduce that $M = \mathsf{mod}_\ell(\mathsf{mod}_\ell(\ldots(\bar{k})))$ for some numeral $k$. Crisply, in order to extract a numeral from $\langle \mu \mid \mathsf{nat} \rangle$ one must evaluate it with more than $\mu$ fuel.

Unlike other canonicity results based on fuel [12], our guarded canonicity result does not depend on a fixed operational semantics or presentation of the equality judgment. The fuel is a purely type-directed notion and throughout the statement and proof of guarded canonicity we manipulate only equivalence classes of terms up to definitional equality.

## 4.1 Proving guarded canonicity

In order to prove Theorem 20, we use a modified variant of *multimodal synthetic Tait computability* [9, 26], a refinement of gluing proofs. Rather than fixing a rewriting system presenting the equational theory of dGuTT, we construct a model of dGuTT lying over the syntactic model comprised essentially of terms equipped with a guarded canonical form. The initiality of syntax then guarantees that every term has a guarded canonical form.

To a first approximation, this proof is a categorical restructuring of a proof-relevant Beth logical relation on equivalences classes of terms up to definitional equality. The predicates associated with each type vary over contexts of the form $\mathbf{0}[\mu].\{\nu\}$, subject to the condition that each predicate collapses to $\{\star\}$ whenever $\mu \leq \nu \circ \xi$ for some $\xi$. Working with proof-relevant predicates enables us to give an elegant interpretation of the universe and manipulating only equivalence classes of terms ensures that we can exploit the various universal properties of connectives like dependent products and sums. We refer the reader to Sterling [26] for a systematic introduction to this style of canonicity proof.

Several caveats complicate this simple picture. In the first instance, it is frequently difficult to construct a model of dGuTT lying over the syntactic model due to the number of strict equations involved. We therefore follow the approach taken by Gratzer [9]: we begin by defining a relaxed category of *Löb cosmoi* and show that there is an appropriate *syntactic Löb cosmos* which enjoys a suitable weakening of initiality. From there, we construct a glued Löb cosmos which supports a rich model of MTT complete with several new primitives. We call the resultant language *multimodal synthetic Tait computability* (MSTC) and use this language to construct the interpretation of types and terms *internally*. In fact, because the model construction takes place within MSTC, once this language is in place the construction of the actual model is *mutatis mutandis* the one given by Gratzer [9].

For the sake of space, therefore, we trace through the aspects of the proof which differ significantly from the normalization proof for MTT. In particular, we show how we construct the glued cosmos necessary to prove guarded canonicity and gloss the interpretation of Löb induction within MSTC. The remaining details – such as the interpretation of all the connectives of MTT – are deferred to Appendices B and C.

▶ **Definition 9.** *A Löb cosmos is a strict 2-functor $G : \mathcal{M} \longrightarrow \mathbf{Cat}$ such that $G(m)$ (which we abusively write $G$) is a locally Cartesian closed category with an initial object and each $G(\mu)$ is a right adjoint. We require the following additional structure:*

1. *A morphism $\tau_G : \dot{\mathcal{T}}_G \longrightarrow \mathcal{T}_G$ in $G$ representing the universe of types and closed under all the connectives of MTT e.g. for each $\mu$ a map $G(\mu)(\mathcal{T}_G) \longrightarrow \mathcal{T}_G$ encoding the formation rule for modal types.*
2. *An element* lob *with the appropriate type and necessary equation.*

*A morphism of Löb cosmoi is a 2-natural transformation of LCC functors satisfying Beck-Chevalley which preserves all structure.*

While Gratzer [9] could organize syntax into a cosmos enjoying a property akin to initiality by taking presheaves on the category of contexts and substitutions $\mathsf{Cx}$, the same maneuver is not available here: the vacuous context $\mathbf{0}$ would no longer be initial when embedded into $\mathbf{PSh}(\mathsf{Cx})$. We therefore localize at $\mathbf{0}_{\mathbf{PSh}(\mathsf{Cx})} \longrightarrow \mathbf{y}(\mathbf{0})$ and consider *sheaves* over the category of contexts. Explicitly, we equip $\mathsf{Cx}$ with a Grothendieck topology $J$:

$$J(\Gamma) = \begin{cases} \{\hom(-,\Gamma), \emptyset\} & \text{if there exists a substitution } \Gamma \longrightarrow \mathbf{0} \\ \{\hom(-,\Gamma)\} & \text{otherwise} \end{cases}$$

The rules in Figure 2 ensure that this Grothendieck topology is subcanonical and that the presheaves of types and terms are in fact sheaves for this topology. It is also easily seen that the precomposition functors induced by $-.\{\mu\}$ restrict to right adjoints on sheaves and that $\mathbf{Sh}(\mathsf{Cx})$ is closed under finite limits and dependent products in $\mathbf{PSh}(\mathsf{Cx})$.

▶ **Theorem 10.** *There is a Löb cosmos $\mathcal{S}(m) = \mathbf{Sh}(\mathsf{Cx})$ where $\mathcal{T}$ (respectively $\dot{\mathcal{T}}$) are realized by the sheaves of types (respectively terms) and $\mathcal{S}(\mu)$ is given by precomposition with $-.\{\mu\}$.*

Passing to sheaves enables us to recover *quasi-projectivity* as described by Gratzer [9]:

▶ **Theorem 11.** *Given a Löb cosmos $G$ and a map $\pi : G \longrightarrow \mathcal{S}$, the following holds:*
1. *For every context $\Gamma\ \mathsf{cx}_{\mathbf{d}}$, there exists an object $[\![\Gamma]\!] : G$ and $\alpha_\Gamma : \pi([\![\Gamma]\!]) \cong \mathbf{y}(\Gamma)$.*
2. *For every type $\Gamma \vdash_{\mathbf{d}} A$, there is a morphism $[\![A]\!] : [\![\Gamma]\!] \longrightarrow \mathcal{T}_G$ such that $\pi([\![A]\!]) \circ \alpha_\Gamma = \lfloor A \rfloor$.*
3. *For every $\Gamma \vdash_{\mathbf{d}} M : A$, there exists $[\![M]\!] : [\![\Gamma]\!] \longrightarrow \dot{\mathcal{T}}_G$ over $[\![A]\!]$ such that $\pi([\![M]\!]) \circ \alpha_\Gamma = \lfloor M \rfloor$.*

*Here $\lfloor - \rfloor$ is half of the isomorphism induced by the Yoneda lemma.*

We can now revise our original proof strategy for guarded canonicity: we will construct a particular Löb cosmos and use Theorem 11 to derive the theorem. We now turn to constructing this Löb cosmos by gluing the syntactic cosmos along a functor to a Grothendieck topos.

As in all gluing proofs, the choice of functor to glue along is crucial. For instance, when proving a standard canonicity result one glues along $\mathbf{PSh}(\mathsf{Cx}) \longrightarrow \mathbf{Set} = \mathbf{PSh}(\mathbf{1})$ given by precomposition with $\mathbf{1} \longrightarrow \mathsf{Cx}$. For normalization, one wishes to work with arbitrary contexts but normal forms are stable under a limited class of *renamings*. Accordingly, one glues along $\mathbf{PSh}(\mathsf{Cx}) \longrightarrow \mathbf{PSh}(\mathsf{Ren})$ given by precomposing with the inclusion $\mathsf{Ren} \longrightarrow \mathsf{Cx}$.

In our case, because we wish to prove a result about terms in context $\mathbf{0}[\mu].\{\nu\}$ we will take a category spanned by contexts of this form. Moreover, because guarded canonical forms are stable under the natural transformations $\mathbf{0}[\mu].\{\nu_0\} \longrightarrow \mathbf{0}[\mu].\{\nu_1\}$, we can recast this subcategory $\mathsf{Cx}$ spanned by contexts $\mathbf{0}[\mu].\{\nu\}$ as a partial order:

▶ **Definition 12.** *Define $(P, \leq)$ to be a partial order whose elements are pairs of modalities $(\mu, \nu)$ such that $(\mu_0, \nu_0) \leq (\mu_1, \nu_1)$ if $\mu_0 = \mu_1$ and $\nu_1 \leq \nu_0$. There is a functor $i : P \longrightarrow \mathsf{Cx}$ sending $(\mu, \nu)$ to $\mathbf{0}[\mu].\{\nu\}$*

Unlike in prior gluing proofs, we represent syntax with $\mathbf{Sh}(\mathsf{Cx})$ rather than $\mathbf{PSh}(\mathsf{Cx})$. Accordingly, we must impose a Grothendieck topology on $P$ so that the inclusion $P \longrightarrow \mathsf{Cx}$ induces a functor $\mathbf{Sh}(\mathsf{Cx}) \longrightarrow \mathbf{Sh}(P)$ and it is this functor that we will glue along.

▶ **Definition 13.** *Transporting the Grothendieck topology on $\mathsf{Cx}$ along the functor $i : P \longrightarrow \mathsf{Cx}$ yields a new topology on $P$ covering $(\mu, \nu)$ with the empty family if $\exists \xi.\ \mu \circ \xi \leq \nu$.*

▶ **Lemma 14.** *Precomposition by $(\mu, \nu) \mapsto (\mu, \nu \circ \xi)$ induces a right adjoint $R_\xi$ : $\mathbf{Sh}(P) \longrightarrow \mathbf{Sh}(P)$.*

▶ **Lemma 15.** *Recalling $\mathcal{S}(\xi) = (-.\{\xi\})^*$ from Theorem 10, $i^* \circ \mathcal{S}(\xi) = R_\xi \circ i^*$.*

Observe that for any pair $(\mu, \nu)$, there exists $n$ such that $\exists \xi.\ \mu \circ \xi \leq \nu \circ \ell^n$. Accordingly, given $X : \mathbf{Sh}(P)$ and $(\mu, \nu) : P$ there exists $n$ such that $R_\ell^n(X)(\mu, \nu) = \{\star\}$. This eventual trivialization ensures that $\mathbf{Sh}(P)$ satisfies Löb induction:

▶ **Lemma 16.** *For any $X : \mathbf{Sh}(P)$ there is a morphism $\mathsf{lob}_X : X^{R_\ell(X)} \longrightarrow X$ satisfying the unfolding equation for Löb induction.*

▶ **Lemma 17.** *Precomposition with $i$ induces a right adjoint $i^* : \mathcal{S} = \mathbf{Sh}(\mathsf{Cx}) \longrightarrow \mathbf{Sh}(P)$.*

Gluing along $i^*$, we obtain a Grothendieck topos $\mathcal{G} = \mathbf{Gl}(i^*)$ whose objects are triples $\big(X : \mathbf{Sh}(\mathsf{Cx}), Y : \mathbf{Sh}(P), f : Y \longrightarrow i^* X\big)$. Intuitively, these are proof-relevant predicates on syntax so that constructing a Löb cosmos in $\mathcal{G}$ is akin to a proof-relevant logical relation.

▶ **Lemma 18.** *There is a strict 2-functor $\mathcal{G} : \mathcal{M} \longrightarrow \mathbf{Cat}$ sending $\mathcal{G}(m) = \mathbf{Gl}(i^*)$ and $\mathcal{G}(\mu)$ is determined component-wise by $\mathcal{S}(\mu)$ and $R_\mu$. Furthermore, $\pi : \mathcal{G} \longrightarrow \mathcal{S}$ is a 2-natural transformation spanned by LCC functors and satisfying Beck-Chevalley.*

**Proof.** This follows from a slight variation on Theorem 4.13 of Gratzer [9] and Lemma 14. ◀

It remains only to equip $\mathcal{G}$ with the structure of a Löb cosmos i.e. a universe $\tau_{\mathcal{G}} : \dot{\mathcal{T}}_{\mathcal{G}} \longrightarrow \mathcal{T}_{\mathcal{G}}$ closed under various connectives. In fact, this process is remarkably routine. $\mathcal{G}$ is a Grothendieck topos and therefore models extensional Martin-Löf type theory with a hierarchy of cumulative universes [11] satisfying the internal realignment principle formulated by Orton and Pitts [21].[2] Moreover, $\mathcal{G}$ is a model of extensional $\mathsf{MTT}$ with a pair of complementary idempotent monads $\bigcirc$ and $\bullet$ presenting $\mathbf{Sh}(\mathsf{Cx})$ (respectively $\mathbf{Sh}(P)$) as an open (resp. closed) subtopos.

This combination of modalities shapes $\mathcal{G}$ into a model of *multimodal synthetic Tait computability*. While there are minor differences in the precise properties of multimodal synthetic Tait computability, this interpretation ensures that we can virtually replay the construction of a universe closed under the connectives of $\mathsf{MTT}$ given by Gratzer [9]. Given this essential similarity, we gloss only Löb induction: the last novelty of this construction.

Interpreting $\mathsf{lob}$ hinges on the fact that $\mathbf{Sh}(P)$ and $R_\ell$ satisfy Löb induction. Lifting Lemma 16 this into the internal language of $\mathcal{G}$ gives us a variant of Löb induction:

$$\textstyle\prod_{A:\mathsf{U}} (\langle \ell \mid \bullet A \rangle \to \bullet A) \to \bullet A$$

This limited constant is sufficient to construct the proper interpretation of Löb induction, which in turn yields the following cousin of the fundamental lemma of logical relations:

▶ **Theorem 19.** *There is a Löb cosmos in $\mathcal{G}$ such that $\pi : \mathcal{G} \longrightarrow \mathcal{S}$ is a map of Löb cosmoi.*

Finally, from Theorems 11 and 19 combined with the definition of terms in $\mathcal{G}$ we conclude:

▶ **Theorem 20** (Guarded canonicity)**.** *Given a term $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} M : A$, the following holds*

- *If $A = \mathsf{nat}$ then there exist a numeral $k$ such that $M = \bar{k}$*
- *If $A = \langle \xi \mid B \rangle$ then $M = \mathsf{mod}_\xi(N)$ for some $\mathbf{0}[\mu].\{\nu \circ \xi\} \vdash_{\mathbf{d}} N : B$*
- *If $A = \mathsf{Id}_B(b_0, b_1)$ then $M = \mathsf{refl}(b_0)$ and $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} b_0 = b_1$ :.*

---

[2] Unlike the well-known construction of universes in presheaf topoi set forth by Hofmann and Streicher [16], Gratzer et al. [11] give a construction which applies in *arbitrary* arbitrary Grothendieck topoi using a small-object argument. This construction requires the axiom of choice and it is currently open whether a constructively acceptable analog exists.

# 5 Proving and programming with guarded streams

While Theorems 5 and 20 ensures that each half of GuTT satisfies one of the two theorems typically used to evaluate usability of a type theory, neither satisfies both. It is unclear, therefore, whether or not sGuTT can really be used to write guarded programs. This is an empirical question, but we argue the affirmative by showing an encoding of synchronous programming [6] in sGuTT and capitalize on Theorems 6 and 20 to compute with the results.

## 5.1 A warmup: defining guarded streams

To begin with, we define the type of guarded streams $\mathtt{Str} : \mathsf{U} \to \mathsf{U}$ in sGuTT:

$$\begin{array}{ll} \mathtt{Str} : \mathsf{U} \to \mathsf{U} & \iota : (A : \mathsf{U}) \to \mathsf{Id}_{\mathsf{U}}(\mathtt{Str}(A), A \times \langle \ell \mid \mathtt{Str}(A) \rangle) \\ \mathtt{Str}(A) = \mathsf{lob}(S.\ A \times \langle \ell \mid S \rangle) & \iota = \mathsf{unfold}(S.\ A \times \langle \ell \mid S \rangle) \end{array}$$

We can define the usual operations on streams using this equivalence. For instance, $\mathtt{cons}$ (respectively $\mathtt{head}$ and $\mathtt{tail}$) is given by transporting along $\iota^{-1}$ (resp. $\iota$). The functoriality of transports then yields an identification $\mathsf{Id}_{\mathtt{Str}(A)}(\mathtt{cons}(\mathtt{head}(s), \mathtt{tail}(s)), s)$.

Fix types $A, B : \mathsf{U}$ and a map $f : A \to B$, we extend this map to a map of streams:

$$\mathtt{map}(f) = \mathsf{lob}(g.\ \lambda s.\ \mathtt{cons}(f(\mathtt{head}(s)), \mathsf{mod}_{\ell}(g) \circledast \mathtt{tail}(s)))$$

▶ **Lemma 21.** *Given $s : \mathtt{Str}(A)$, there is an identification $\mathsf{Id}_{\mathtt{Str}(A)}(\mathtt{map}(\mathsf{id}, s), s)$.*

**Proof.** We prove this by Löb induction so we are given an induction hypothesis:

$$r : \left(\ell \,\middle|\, (s : \mathtt{Str}(A)) \to \mathsf{Id}_{\mathtt{Str}(A)}(\mathtt{map}(\mathsf{id}, s), s)\right)$$

Now fix $s : \mathtt{Str}(A)$. We have an identification between $s$ and $\mathtt{cons}(\mathtt{head}(s), \mathtt{tail}(s))$. The left-hand side is identified with $\mathtt{cons}(\mathsf{id}(\mathtt{head}(s)), \mathsf{next}(\mathtt{map}(\mathsf{id})) \circledast \mathtt{tail}(s))$ via $\mathsf{unfold}(\dots)$. It then suffices to construct an element of $\mathsf{Id}_{\langle \ell | \mathtt{Str}(A) \rangle}(\mathsf{next}(\mathtt{map}(\mathsf{id})) \circledast \mathtt{tail}(s), \mathtt{tail}(s))$. To this end, we use the elimination rule for modal types: Applying this rule to $\mathtt{tail}(s)$, it suffices to construct the following identification for an arbitrary $x : (\ell \mid \mathtt{Str}(A))$:

$$\mathsf{Id}_{\langle \ell | \mathtt{Str}(A) \rangle}(\mathsf{mod}_{\ell}(\mathtt{map}(\mathsf{id}, x)), \mathsf{mod}_{\ell}(x))$$

The result is now an immediate consequence of $r(x)$ combined with Equation (3). ◀

▶ **Lemma 22.** *Given functions $f : A \to B$, $g : B \to C$, for each $s : \mathtt{Str}(A)$ there is an identification $\mathsf{Id}_{\mathtt{Str}(C)}(\mathtt{map}(g \circ f, s), \mathtt{map}(g, \mathtt{map}(f, s)))$.*

The proofs of the above lemmas (see the one for Lemma 21), show how several features of sGuTT are crucial for programming. In particular, we have made heavy use of the *propositional* unfolding rules for Löb induction to manipulate a definition using lob, and we rely on crisp identity induction principles to use Löb induction to prove an equality.

## 5.2 A logical foundation for synchronous programming

In order to test the limits of sGuTT, we revisit the abstract encoding of synchronous programming [14] (as implemented by e.g. Lustre [13]) in dependent type theory with coinductive streams given by Boulmé and Hamon [6]. We repeat their construction with the guarded streams defined in Section 5.1. Intuitively, we interpret a type $\tau$ of their calculus as

guarded stream of values in GuTT. By using guarded streams, however, we are able to simplify the construction of this model and eliminate the need to manually ensure well-formedness of various constructions. Moreover, we show that propositional Löb is sufficient for guarded programming.

We first define a type of clocks or warps which dictate the rate at which a stream is expected to produce values. Following Boulmé and Hamon [6], these are realized by streams of booleans $W = \mathtt{Str}(\mathsf{bool})$. The primitive objects of our encoding are specialized streams indexed by a warp:

$$\mathtt{WStr}(A) = \mathsf{lob}(B.\ \lambda w.\ \mathsf{let}\ \mathsf{mod}_\mu(u) \leftarrow \mathtt{tail}(w)\ \mathsf{in}\ \mathsf{if}\ \mathtt{head}(w)\ \mathsf{then}\ A \times \langle \ell \mid B(u) \rangle\ \mathsf{else}\ \langle \ell \mid B(u) \rangle)$$

So a WStr is indexed by a type $A$ and warp $w$, and contains values of $A$ only when $w$ is true. This is similar to the model of Lustre detailed by Boulmé and Hamon [6], but in contrast to *loc. cit.*, which uses coinductive streams, our use of guarded streams facilitates a number of conceptual simplifications while immediately guaranteeing some theorems.

Henceforth we will stop explicitly using lob to define guarded recursive functions, and adopt the more typical informal style so that the above definition could be written

$$\mathtt{WStr}(A, \mathsf{cons}(b, \mathsf{mod}_\ell(w))) = \mathsf{if}\ b\ \mathsf{then}\ A \times \langle \ell \mid \mathtt{WStr}(A, w) \rangle\ \mathsf{else}\ \langle \ell \mid \mathtt{WStr}(A, w) \rangle$$

We also adopt pattern-matching syntax for modal types and equivalences like cons as well as a **rewrite** based on the available operation in Agda. We further avail ourselves of cons, head, and tail for WStr. These have slightly unusual types to account for the presence of warps and the potential absence of data at a given step.

$$\begin{aligned}
&\mathtt{cons} : (b : \mathsf{bool})(\ell \mid w : W) \\
&\qquad \to (\mathsf{if}\ b\ \mathsf{then}\ A \times \langle \ell \mid \mathtt{WStr}(A, w) \rangle\ \mathsf{else}\ \langle \ell \mid \mathtt{WStr}(A, w) \rangle) \to \mathtt{WStr}(A, \mathsf{cons}(b, \mathsf{mod}_\ell(w))) \\
&\mathtt{head} : (\ell \mid w : W) \to \mathtt{WStr}(A, \mathsf{cons}(\mathsf{tt}, \mathsf{mod}_\ell(w))) \to A \\
&\mathtt{tail} : (b : \mathsf{bool})(\ell \mid w : W) \to \mathtt{WStr}(A, \mathsf{cons}(b, \mathsf{mod}_\ell(w))) \to \langle \ell \mid \mathtt{WStr}(A, w) \rangle
\end{aligned}$$

We use this new surface syntax to define the constant stream of values:

$$\begin{aligned}
&\mathtt{const} : A \to (w : W) \to \mathtt{WStr}(A, w) \\
&\mathtt{const}(x, \mathsf{cons}(b, \mathsf{mod}_\ell(w))) = \mathsf{if}\ b\ \mathsf{then}\ \mathsf{cons}(x, \mathsf{mod}_\ell(\mathtt{const}(x, w)))\ \mathsf{else}\ \mathsf{cons}(\mathsf{mod}_\ell(\mathtt{const}(x, w)))
\end{aligned}$$

Next, we define a combinator allowing us to zip two streams and combine their values:

$$\begin{aligned}
&\mathtt{zip} : (w : W) \to \mathtt{WStr}(A \to B, w) \to \mathtt{WStr}(A, w) \to \mathtt{WStr}(B) \\
&\mathtt{zip}(\mathsf{cons}(b, \mathsf{mod}_\ell(w)), s_1, s_2) = \\
&\quad \mathsf{if}\ b \\
&\quad \mathsf{then}\ \mathsf{cons}(\mathtt{head}(s_1)(\mathtt{head}(s_2)), \mathsf{mod}_\ell(\mathtt{zip}(w)) \circledast \mathtt{tail}(s_1) \circledast \mathtt{tail}(s_2)) \\
&\quad \mathsf{else}\ \mathsf{cons}(\mathsf{mod}_\ell(\mathtt{zip}(w)) \circledast \mathtt{tail}(s_1) \circledast \mathtt{tail}(s_2))
\end{aligned}$$

While we omit the definitions, we also can define a variant of Lustre's fby operator, which delays a stream by one and replaces the first element:

$$\mathtt{fby} : (w : W) \to A \to \mathtt{WStr}(A, \mathsf{cons}(\mathsf{ff}, \mathsf{mod}_\ell(w))) \to \mathtt{WStr}(A, \mathsf{cons}(\mathsf{tt}, \mathsf{mod}_\ell(w)))$$

In fact, we have provided definitions of all the combinators presented by Boulmé and Hamon [6], including several which further exercise the dependent types of sGuTT.

## 5.3 Running an example program

Thus far we have carefully worked in sGuTT, so our programs can be type-checked. Unfortunately, the lack of canonicity in sGuTT means that they cannot be run. The split nature of GuTT, however, guarantees equivalent programs in dGuTT which can be executed. To concretize this discussion, consider the following stream of natural numbers in sGuTT:

$$\begin{aligned}&\texttt{nats} : \texttt{WStr}(\mathsf{nat}, \mathsf{lob}(\texttt{cons}(\mathsf{tt}, \mathsf{mod}_\ell(-))))\\&\texttt{nats}\ \textbf{rewrite}\ \mathsf{unfold}(\dots) = \texttt{fby}(0, \texttt{zip}(\texttt{cons}(\mathsf{next}(\texttt{const}(\lambda n.\ \mathsf{suc}(n)))), \texttt{cons}(\mathsf{mod}_\ell(\texttt{nats}))))\end{aligned}$$

While this program is relatively simple, we can already detect something unsatisfactory: it is hard to verify that the second element of this stream is 1. Using Theorem 6, we interpret nats as a program in dGuTT where the corresponding equality follows from computation:

$$\begin{aligned}\mathsf{next}(\texttt{head}) \circledast \texttt{tail}(\texttt{nats}) &= \mathsf{next}(\texttt{head}) \circledast \mathsf{mod}_\ell(\texttt{zip}(\texttt{const}(\lambda n.\ \mathsf{suc}(n))), \texttt{nats})\\&= \mathsf{mod}_\ell(\texttt{head}(\texttt{zip}(\texttt{const}(\lambda n.\ \mathsf{suc}(n))), \texttt{nats}))\\&= \mathsf{mod}_\ell(1)\end{aligned}$$

These equalities are definitional; in dGuTT our stream combinators reduce on-the-nose. Moreover, Theorem 7 ensures that if we were to interpret the original sGuTT program into e.g., the topos of trees, the second element would indeed be 1.

This example is indicative of a general phenomenon: we can always program in sGuTT to type-check a program and switch to dGuTT to execute it and obtain a concrete result. Theorem 20 guarantees that this always yields a numeral, and Theorem 7 ensures that we obtain the expected results in standard models of sGuTT such as the topos of trees.

# 6 Related work

GuTT is closely related to a number of modal and guarded type theories. Most directly, GuTT refines guarded recursion in MTT and improves upon the proposed guarded MTT presented by Gratzer et al. [10] by avoiding equality reflection. By replacing this rule with propositional Löb induction as well as a crisp identity induction principle, the theory remains practicable. Moreover, by extending with only these principles we are able to benefit from the metatheory of MTT and its prototype proof assistant, which implements sGuTT [25].

### Clocked Type Theory

Clocked Type Theory (CloTT) [2,17,18] is a family of closely related guarded type theories which also rely on dependent right adjoints to structure the later modality. Like sGuTT, CloTT escapes Corollary 3 by preventing Löb induction in most circumstances. Rather than having a separate type theory where Löb induction does compute, CloTT carefully allows lob to unfold only when at the "top-level". In particular, if we were to construct the gadget used by Corollary 3 in CloTT, only the outer application of Löb would unfold.

Unfortunately, the additional definitional equality provided CloTT cannot be used in most circumstances; whenever a user is manipulating terms in a non-trivial context, Löb induction unfolds only up to a propositional equality, just as with sGuTT. Moreover, in the existing formulation of CloTT in Agda [27], Löb induction is added as an *axiom* without even the definitional equality unfolding it at the top-level. The fact that CloTT has been used to formalize substantial arguments under these circumstances serves as further evidence for the practicability of propositional Löb induction as available in sGuTT.

Accordingly, while we conjecture that it is possible to add a similar limited form of unfolding for Löb induction to sGuTT, this would not impact the use of the calculus. For example, in Section 5.2 we could not benefit from the addition of these definitional equalities; we could only take advantage of them when calculating a closed program as in Section 5.3 and at the cost of adding other machinery to the theory to ensure that congruence does not allow the unfolding of Löb to escape to arbitrary positions in the program.

By avoiding this top-level unfolding, GuTT is able to take advantage of results about MTT off-the-shelf. Consequently, the metatheory of GuTT is more developed than CloTT; the decidability of type-checking of GuTT is proven – with a prototype implementation [25] – and an appropriate canonicity theorem is proven for the full system. Both of these results are conjectured for CloTT, but neither have been proven for a complete version of the system.

### Synthetic Tait computability

We have proven Theorem 20 by adapting (multimodal) synthetic Tait computability (STC) [9, 26]. This is the first application of STC – multimodal or not – which takes full advantage of the fact that STC applies to arbitrary sheaf topoi rather than just presheaf topoi. Moreover, this result expands the reach of STC to yet another class of languages: guarded recursion. Theorem 20 therefore opens the door to applying STC to guarded programming languages.

## 7 Conclusions

We contribute GuTT, a type theory for guarded recursion featuring a novel decomposition into "static" and "dynamic" fragments. We prove that the static fragment enjoys normalization and decidability of type-checking and formulate and prove a guarded canonicity result for the "dynamic" fragment. In so doing, we escape a no-go theorem (Corollary 3) showing that the presence of Löb induction is all but certain to disrupt the decidability of conversion. Finally, we have shown that this dichotomy between static and dynamic results in a usable system by carrying out a case-study based on synchronous programming.

In the future, we hope to further develop a prototype implementation of GuTT based on the work of Stassen et al. [25] and potentially extend GuTT with a richer mode theory for guarded recursion to permit more flexible guarded programming.

We also intend to investigate whether sGuTT enjoys a "homotopy canonicity" property i.e., while not every $M : \mathsf{nat}$ is definitional equal to a numeral, we conjecture that $M$ is *propositionally* equal to a numeral $\bar{n}$. If homotopy canonicity holds, then $\bar{n}$ is necessarily definitionally equal to the numeral yielded by Theorem 20. More generally, we conjecture that dGuTT is conservative over sGuTT. Such a result would diminish some of the importance of dGuTT, but the dynamic portion of the theory would still serve as an important computational justification for sGuTT and will almost certainly still prove more convenient when using GuTT as an internal language.[3]

[3] One may fruitfully compare this to the situation with intensional and extensional type theory. Hofmann's celebrated conservativity result [15] has not obviated extensional type theory, though it has clarified the relationship between the two theories.

## References


1. Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 197–208. Association for Computing Machinery, 2013. `doi:10.1145/2500365.2500597`.
2. Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017. `doi:10.1109/LICS.2017.8005097`.
3. Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. `doi:10.1017/S0960129519000197`.
4. Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. `doi:10.2168/LMCS-8(4:1)2012`.
5. Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. Guarded Dependent Type Theory with Coinductive Types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 20–35. Springer Berlin Heidelberg, 2016.
6. Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 495–506, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
7. Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 407–421. Springer Berlin Heidelberg, 2015.
8. Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 120–134, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. `doi:10.1007/3-540-61780-9_66`.
9. Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. `doi:10.1145/3531130.3532398`.
10. Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. `doi:10.46298/lmcs-17(3:11)2021`.
11. Daniel Gratzer, Michael Shulman, and Jonathan Sterling. Strict universes for Grothendieck topoi, 2022. `arXiv:2202.12012`.
12. Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18. ACM, 2018. `doi:10.1145/3209108.3209148`.
13. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. `doi:10.1109/5.97300`.
14. Nicolas Halbwachs. Synchronous programming of reactive systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 1–16, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
15. Martin Hofmann. Conservativity of equality reflection over intensional type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 153–164, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
16. Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished note, 1997. URL: `https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf`.
17. Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks, 2021. `arXiv:2102.01969`.

18 Bassel Mannaa, Rasmus Ejlers Møgelberg, and Niccolò Veltri. Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory. *Logical Methods in Computer Science*, Volume 16, Issue 4, December 2020. doi:10.23638/LMCS-16(4:17)2020.

19 Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, 2014. doi:10.1145/2603088.2603132.

20 H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266. IEEE Computer Society, 2000.

21 Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science*, 14(4), 2018. doi:10.23638/LMCS-14(4:23)2018.

22 François Pottier. A typed store-passing translation for general references. In *ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, January 2011. doi:10.1145/1925844.1926403.

23 Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory. *Logical Methods in Computer Science*, 16(1), 2020. arXiv:1706.07526.

24 Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 25(5):1203–1277, 2015. doi:10.1017/S0960129514000565.

25 Philipp Stassen, Daniel Gratzer, and Lars Birkedal. A flexible multimodal proof assistant, 2022. To appear in Workshop on the Implementation of Type Systems 2022.

26 Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, 2021. CMU technical report CMU-CS-21-142. doi:10.5281/zenodo.5709838.

27 Niccolò Veltri and Andrea Vezzosi. Formalizing $\pi$-calculus in guarded cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 270–283, 2020.


# A Models of GuTT

A model of sGuTT extends a model of MTT instantiated with $\mathcal{M}$ with two additional constants for lob and unfold.

▶ **Definition 23.** *A model of MTT with mode $\mathcal{M}$ consists of a 2-functor $F : \mathcal{M}^{\mathsf{coop}} \longrightarrow \mathbf{Cat}$ such that $F(m)$ has a terminal object along with the following structure:*

- *A* modal *representable natural transformation $\tau : \dot{\mathcal{T}} \longrightarrow \mathcal{T}$ [10].*
- *A choice of codes closing $\tau$ under dependent sums, modal dependent products, natural numbers, modalities, etc.*
- *A choice of code closing $\tau$ under identity types and left-lifting structure further equipping those types with a crisp induction principle.*

▶ **Definition 24.** *A model of sGuTT consists of a model of MTT further supplemented with two elements of $\dot{\mathcal{T}}$ witnessing* lob *and* unfold*.*

▶ **Definition 25.** *A model of dGuTT consists of a model of sGuTT satisfying the following conditions:*

- lob *satisfies the expected definitional equality*
- unfold *is precisely interpreted by reflexivity*
- *There is a chosen initial object $\mathbf{0}$ in $F(m)$ which is preserved by each $F(\mu)$*
- $\dot{\mathcal{T}}(\mathbf{0}) = \mathcal{T}(\mathbf{0}) = \{\star\}$
- *A choice of objects representing the functors $\hom(-.\{\mu\}, \mathbf{0})$ for each $\mu$.*

# B Löb cosmoi

Here we give a more precise definition of Löb cosmoi. We presuppose knowledge with Section 3 of Gratzer [9] which introduces the notion of *MTT cosmoi*.

▶ **Definition 26.** *A Löb cosmos is a strict 2-functor $G : \mathcal{M} \longrightarrow \mathbf{Cat}$ such that $G(m)$ (which we abusively write $G$) is a locally Cartesian closed category with an initial object and each $G(\mu)$ is a right adjoint. We require the following additional structure:*

1. *A morphism $\tau_G : \dot{\mathcal{T}}_G \longrightarrow \mathcal{T}_G$ in $G$*
2. *A choice of codes ensuring that $\tau_G$ is closed under dependent sums, weak natural numbers, identity types with a crisp induction principle and a weak Tarski universe. The last is itself closed under all connectives except the universe.*
3. *A choice of code making $\tau_G$ closed under modal dependent products: a code for $\mathbf{P}_{G(\mu)\tau_G}(\tau_G)$ for each $\mu : m \longrightarrow m$.*
4. *An element* lob *with the appropriate type and necessary equation.*
5. *For each $\mu$, there exists a chosen commuting square*

$$\begin{array}{ccc} G(\mu)(\dot{\mathcal{T}}_n) & \longrightarrow & \dot{\mathcal{T}}_m \\ \downarrow & & \downarrow \\ G(\mu)(\mathcal{T}_n) & \longrightarrow & \mathcal{T}_m \end{array} \tag{4}$$

6. *For each $\mu : m \longrightarrow m$ and $\nu : m \longrightarrow m$, there is a chosen lifting structure $G(\mu)(m) \pitchfork G(\mu \circ \nu)(\mathcal{T}_G) \times \tau_G$, where $m : G(\nu)(\dot{\mathcal{T}}_o) \longrightarrow G(\nu)(\mathcal{T}_G) \times_{\mathcal{T}_G} \dot{\mathcal{T}}_G$ is the comparison map induced by Diagram 4.*

▶ **Definition 27.** *A morphism of Löb cosmoi is a 2-natural transformation whose components are LCC functors preserving the initial object up to isomorphism and all structure strictly.*

We can form a Löb cosmos in $\mathbf{Sh}(\mathsf{Cx})$: sheaves over contexts and substitutions localizing the map $\mathbf{0}_{\mathbf{PSh}(\mathsf{Cx})} \longrightarrow \mathbf{y}(\mathbf{0})$.

▶ **Theorem 28.** *This topology is subcanonical.*

The universe of types and terms is taken is given by the standard representable universe (it is a map of sheaves by the $\eta$-laws governing $\mathbf{0}$) which we write $\tau_{\mathbf{Sh}(\mathsf{Cx})}$:

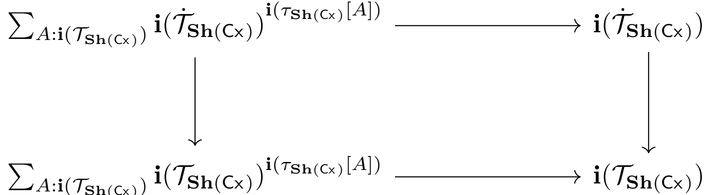
$$\mathcal{T}_{\mathbf{Sh}(\mathsf{Cx})}(\Gamma) = \{A \mid \Gamma \vdash_{\mathbf{d}} A\}$$
$$\dot{\mathcal{T}}_{\mathbf{Sh}(\mathsf{Cx})}(\Gamma) = \{(A, M) \mid \Gamma \vdash_{\mathbf{d}} M : A\}$$

We must show that $\tau$ is closed under the various required connectives. These proofs are based on the following observation.

▶ **Lemma 29.** *Viewed as a subcategory of $\mathbf{PSh}(\mathsf{Cx})$, $\mathbf{Sh}(\mathsf{Cx})$ is closed under limits and dependent products.*

▶ Remark 30. A type-theoretic proof of this the second half of this lemma follows from Lemma 1.26 of Rijke et al. [23].

We already know that $\tau$ is closed under dependent sums and products in $\mathbf{PSh}(\mathsf{Cx})$ so that there is a Cartesian square



However, sheaves are closed under the formation of finite limits and dependent products, so this is already a Cartesian square in $\mathbf{Sh}(\mathsf{Cx})$. A similar argument holds for dependent sums and natural numbers.

It remains to describe the interpretations of modalities on this category. Given a modality $\mu$ we define $F_\mu = (-.\{\mu\})^*$. We first must show that this sends sheaves to sheaves: there is an isomorphism $\mathbf{0}.\{\mu\} \cong \mathbf{0}$ so if $\Gamma$ is covered by the empty sieve so too is $\Gamma.\{\mu\}$. This functor also preserves all limits (they are computed pointwise in sheaves) so it is a right adjoint. We must show that it is weakly representable, but we already have such a representation in presheaves and it again descends.

▶ **Theorem 31.** *There is a Löb cosmos $\mathcal{S}$ sending $m$ to $\mathbf{Sh}(\mathsf{Cx})$ and each modality $\mu$ to $F_\mu$. The universe of types and terms is interpreted by $\tau$ and all the connectives descend from $\mathbf{PSh}(\mathsf{Cx})$ in the manner described above.*

## C The canonicity cosmos

In this appendix we highlight the novel constructions of the canonicity cosmos. As with Appendix B, we again presuppose some familiarity with Gratzer [9]. Specifically, we make use of the internal presentation of cosmoi discussed in Section 3, the language of multimodal synthetic Tait computability introduced in Section 4, and the construction of the normalization cosmos from Section 5.

As with the construction of the normalization cosmos of Gratzer [9], we now construct a Löb cosmos in $\mathcal{G}$ which lies strictly over the syntactic Löb cosmos $\mathcal{S}$. We proceed internally to $\mathcal{G}$, using the language of multimodal synthetic Tait computability to define a sequence of constants.

▶ **Definition 32.** *The language of multimodal synthetic Tait computability consists of extensional MTT together with a universe of strict propositions, a distinguished proposition* $\mathbf{syn} : \Omega$ *inducing a pair of complementary lex idempotent monads $\bigcirc = \mathbf{syn} \to -$ and $\bullet = \mathbf{syn} \vee -$. Furthermore, each universe of MTT satisfies the internal realignment axiom of Orton and Pitts [21] and both lex monads commute with $\langle \mu \mid - \rangle$.*

A word of caution is required as – unlike the situation described by Grater [9] – $\langle \mu \mid \mathbf{syn} \rangle \neq \mathbf{syn}$. This divergence stems essentially from the fact that $\langle \ell \mid - \rangle$ does not preserve $\mathbf{0}$. However, this is immaterial as $\langle \mu \mid - \rangle$ does commute with both $\bigcirc$ and $\bullet$.

When interpreted into $\mathcal{G}$, the $\bigcirc$-modal correspond to objects in $\mathcal{S}$. Accordingly, the syntactic model is manifested as a series of $\bigcirc$-modal types ($\mathsf{Ty}$, $\mathsf{Tm}$, etc.) and constants as done by Gratzer [9]. We then must construct a series of types and constants in MTT which collapse to the syntactic model under $\mathbf{syn}$.

▶ Remark 33. We write $\bigcirc_z A(z)$ as shorthand for the dependent open modality $\prod_{z:\mathbf{syn}} A(z)$.

▶ Remark 34. We write $\{A \mid z : \phi \mapsto a(z)\}$ for the type of elements of $A$ which are equal to $a(z)$ under the assumption $z : \phi$. We treat the coercion to $A$ as silent.

▶ Remark 35. We will write `foo`* for the constant lying over `foo`.

We begin by defining the following type using realignment to ensure that it lies over $\mathsf{Ty}$:

$$\begin{aligned}&\textbf{record } \mathsf{Ty}^* : \{\mathsf{U}_2 \mid z : \textbf{syn} \mapsto \mathsf{Ty}(z)\} \textbf{ where}\\&\quad \mathsf{code} : \bigcirc_z \mathsf{Ty}(z)\\&\quad \mathsf{pred} : \{\mathsf{U}_1 \mid z : \textbf{syn} \mapsto \mathsf{Tm}(z, \mathsf{code})\}\end{aligned} \tag{5}$$

We further define $\mathsf{Tm}^* : \{\mathsf{Ty}^* \to \mathsf{U}_1 \mid z : \textbf{syn} \mapsto \mathsf{Tm}(z)\}$ to send $A$ to $\mathsf{pred}(A)$.

The following lemma follows more-or-less verbatim the proofs given by Gratzer [9].

▶ **Lemma 36.** *The gluing model supports dependent products and sums, a weak Tarski universe, identity types with a crisp induction principle, and modal types.*

▶ **Lemma 37.** *The gluing model supports weak natural numbers.*

**Proof.** We begin by defining $\mathsf{Nat}^*$:

$$\begin{aligned}&\mathsf{code}(\mathsf{Nat}^*) = \mathsf{Nat}\\&\mathsf{pred}(\mathsf{Nat}^*) = \textstyle\sum_{M:\bigcirc_z \mathsf{Nat}(z)} \bullet\left[\sum_{n:\mathsf{nat}} \bar{n} = M\right]\end{aligned}$$

The two introduction forms deviate only slightly from the standard definitions:

$$\begin{aligned}&\mathsf{zero}^* = (\mathsf{zero}, \eta(0, \star))\\&\mathsf{succ}^* = \lambda M.\ (\mathsf{succ}(M), (n, \star) \leftarrow \pi_1(M); \eta(n+1, \star))\end{aligned}$$

We finally define $\mathsf{rec}^*$:

$$\mathsf{rec}^*(A, M_z, M_s, N) = \begin{cases}\mathsf{rec}^*(z, M_z, M_s, N) & \pi_1(N) = \iota_0(z)\\ M_s^n(M_z) & \pi_1(N) = \iota_1(n, \star)\end{cases}$$ ◀

▶ **Lemma 38.** *There is a constant* $\mathsf{lob}^* : (\langle \ell \mid \mathsf{Tm}^*(A)\rangle \to \mathsf{Tm}^*(A)) \to \mathsf{Tm}^*(A)$ *lying strictly over* $\mathsf{lob}$ *and satisfying the unrolling rule.*

**Proof.** Let us fix $f : \langle \ell \mid \mathsf{Tm}^*(A)\rangle \to \mathsf{Tm}^*(A)$. We must construct $\mathsf{lob}^*(f)$. We now use the fracture theorem: $A \cong \bigcirc A \times_{\bullet\bigcirc A} \bullet A$.

We have the left component of this pullback already: $\lambda z.\ \mathsf{lob}(z, f)$. It remains to construct an element of $\bullet\mathsf{Tm}^*(A)$ which coheres appropriately with $\mathsf{lob}$. Here we use $\mathsf{lob}_\bullet$ induction with the target $\bullet\mathsf{Tm}^*(A)$. We must produce a function $\langle \ell \mid \bullet\mathsf{Tm}^*(A)\rangle \to \bullet\mathsf{Tm}^*(A)$. We have $g = \bullet f : \langle \ell \mid \bullet A\rangle \longrightarrow \bullet A$. Therefore $\mathsf{lob}_\bullet(g) : \bullet\mathsf{Tm}^*(A)$. We must show the following:

$$(\bullet\eta_\bigcirc)(\mathsf{lob}_\bullet(g)) = \eta_\bullet(\lambda z.\ \mathsf{lob}(z, f)) \tag{6}$$

Let us prove this through Löb induction, available because equality of terms of $\bullet$-modal type is a $\bullet$-modal type. We then assume Equation (6) under $\langle \ell \mid -\rangle$. Taking advantage of $\langle \ell \mid -\rangle$ as a fully-fledged dependent right adjoint, we rephrase this assumption as the equality

$$\mathsf{next}(\bullet\eta_\bigcirc(\mathsf{lob}_\bullet(g))) = \mathsf{next}(\eta_\bullet(\lambda z.\ \mathsf{lob}(z, f)))$$

We may simplify this by taking advantage of our ability to commute MTT modalities past those induced by gluing:

$$\bullet(\eta_\bigcirc \circ \mathsf{next})(\mathsf{lob}_\bullet(g)) = \eta_\bullet(\lambda z.\ \mathsf{next}(\mathsf{lob}(z, f))) : \bullet\bigcirc\langle \ell \mid \mathsf{Tm}^*(A)\rangle$$

Returning now to our goal, after applying both computation rules for the different forms of Löb induction, we are left with the following:

$$\bullet\eta_{\bigcirc}(g((\bullet\mathsf{next})(\mathsf{lob}_{\bullet}(g)))) = \eta_{\bullet}(\lambda z.\ f(\mathsf{next}(\mathsf{lob}(z,f))))$$

Rewriting, we obtain

$$\bullet(\bigcirc f \circ \eta_{\bigcirc} \circ \mathsf{next})(\mathsf{lob}_{\bullet}(g)) = (\bullet\bigcirc f)(\eta_{\bullet}\lambda z.\ \mathsf{next}(\mathsf{lob}(z,f)))$$

The result now follows from our induction hypothesis. ◀

▶ **Theorem 39.** *$\mathcal{G}$ supports a Löb cosmos equipped with a projection onto $\mathcal{S}$.*

▶ **Theorem 20** (Guarded canonicity)**.** *Given a term $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} M : A$, the following holds*

- *If $A = \mathsf{nat}$ then there exist a numeral $k$ such that $M = \bar{k}$*
- *If $A = \langle \xi \mid B \rangle$ then $M = \mathsf{mod}_{\xi}(N)$ for some $\mathbf{0}[\mu].\{\nu \circ \xi\} \vdash_{\mathbf{d}} N : B$*
- *If $A = \mathsf{Id}_B(b_0, b_1)$ then $M = \mathsf{refl}(b_0)$ and $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} b_0 = b_1$ :.*

**Proof.** Fix a term $\mathbf{0}[\mu].\{\nu\} \vdash_{\mathbf{d}} M : A$. By Theorem 11 we obtain an element of $M^* : A^*$ which lies over $M$ up to isomorphism of contexts. We have the following square in $\mathbf{Sh}(P)$

$$\begin{array}{ccc} (F_\mu)_!(F_\nu(\mathbf{0}_{\mathbf{Sh}(P)})) & \longrightarrow & (\tau_{\mathcal{G}}[A^*])_0 \\ \downarrow & & \downarrow \tau_{\mathcal{G}}[A^*] \\ i^*\mathbf{y}(\mathbf{0}[\mu].\{\nu\}) & \xrightarrow[i^*\mathbf{y}(M)]{} & i^*(\tau_{\mathcal{S}}[A]) \end{array} \tag{7}$$

We instantiate this diagram of sheaves at $(\mu, \nu)$. Let us construct an $(\mu, \nu)$-point of $(F_\nu)_!(F_\mu(\mathbf{0}_{\mathbf{Sh}(P)}))$. By functoriality, it suffices to construct a $(\mu, \mathsf{id})$ point of $F_\mu(\mathbf{0})$. Transposing, it is sufficient to construct a $(\mu, \mu)$-point of $\mathbf{0}$, which exists uniquely (it is a map between initial objects). Calculating, this lies over $\mathsf{id} : i^*\mathbf{y}(\mathbf{0}[\mu].\{\nu\})(\mu, \nu)$. This, together with the definition of the computability data for $\mathsf{nat}$, $\langle - \mid - \rangle$, and $\mathsf{Id}_{-}(-,-)$, yields the desired result. ◀

# Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity

**Frédéric Blanqui**

Université Paris-Saclay, INRIA, ENS Paris-Saclay, CNRS, Laboratoire Méthodes Formelles, 4 avenue des Sciences 91190 Gif-sur-Yvette, France

**Abstract**

The encoding of proof systems and type theories in logical frameworks is key to allow the translation of proofs from one system to the other. The $\lambda\Pi$-calculus modulo rewriting is a powerful logical framework in which various systems have already been encoded, including type systems with an infinite hierarchy of type universes equipped with a unary successor operator and a binary max operator: Matita, Coq, Agda and Lean. However, to decide the word problem in this max-successor algebra, all the encodings proposed so far use rewriting with matching modulo associativity and commutativity (AC), which is of high complexity and difficult to integrate in usual algorithms for $\beta$-reduction and type-checking. In this paper, we show that we do not need matching modulo AC by enforcing terms to be in some special canonical form wrt associativity and commutativity, and by using rewriting rules taking advantage of this canonical form. This work has been implemented in the proof assistant Lambdapi.

**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, type theory, type universes, rewriting

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.24

**Supplementary Material** *Software (Source Code)*: `https://github.com/Deducteam/lambdapi/blob/master/src/core/term.ml`
archived at `swh:1:cnt:4fe22b374435a880877522ae26106c089df55178`

**Acknowledgements** The author thanks Thiago Felicissimo for his testing and remarks on the implementation of the present work in `https://github.com/Deducteam/lambdapi`, Guillaume Genestier for his careful reading of a first version of this paper, Gaspard Férey for his remarks on a first version of this paper, as well as the anonymous reviewers for their suggestions.

## 1 Introduction

The complete formalization of important mathematical theorems or software is possible but still very costly in terms of time and expertise (seL4, compcert, odd-order theorem, etc.). Moreover, all these certifications are specific to a given prover, and rely on its implementation and maintenance. And it is currently very difficult to automatically translate developments done in one system to another system, especially if those systems are based on different, and possibly incompatible, foundations. Hence, there is a lot of work duplication, and it gets more and more difficult for new proof systems to emerge as the development of standard libraries is time-consuming and not very rewarding.

**Logical frameworks.** A way to improve this situation is to encode the axioms and rules of proof systems into a common language, called a logical framework, so that a feature (e.g. polymorphism) that is common to two different systems is encoded by the same construction [10]. Using a logical framework for $n$ systems allows one to reduce the number of translators necessary to translate each system to all the others from $\mathcal{O}(n^2)$ to $\mathcal{O}(2n)$.

© Frédéric Blanqui;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 24; pp. 24:1–24:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The $\lambda\Pi$-calculus modulo rewriting, $\lambda\Pi/\mathcal{R}$, is a good candidate for such a logical framework [10]. In [14] already, Cousineau and Dowek proved that any functional pure type system (PTS) [7] can be encoded in $\lambda\Pi/\mathcal{R}$. Then, several other systems have been encoded: higher-order logic and the OpenTheory format used by HOL-Light and HOL4, the calculus of inductive constructions and the proof systems of Matita [5], Coq [17] and Agda [19].

$\lambda\Pi/\mathcal{R}$ extends the logical framework LF [22] by allowing the definition of function symbols and types by a set $\mathcal{R}$ of rewriting rules [34]. LF itself extends Church's simply-typed $\lambda$-calculus with dependent types, that is, object-indexed type families. Given a type $A$, written $A : \mathrm{Type}$, the product of an $A$-indexed family of types $(B(x))_{x\in A}$ is written $\Pi x : A, B(x)$, and simply $A \to B$ if $B(x)$ does not depend on $x$. In LF, types equivalent modulo $\beta$-conversion are identified while, in $\lambda\Pi/\mathcal{R}$, types equivalent modulo $\beta\mathcal{R}$-conversion are identified.

For the type conversion and type-checking of $\lambda\Pi/\mathcal{R}$ to be decidable, one usually requires the rewrite relation generated by $\beta$-reduction and rewrite rules, $\longrightarrow_\beta \cup \longrightarrow_\mathcal{R}$, to preserve typing, be confluent (the order of reductions does not matter) and terminating (there is no infinite rewrite sequence), and various criteria have been developed to check those properties (see for instance [9, 19, 17]).

**Type universes** are a way to reify types, that is, to see types as objects [28], which allows one to express polymorphism (quantification over types) and build models of type theory in type theory, like in set theory inaccessible cardinals allow one to build models of ZF. By iterating this process, we get an $\omega$-indexed sequence of type-theoretic universes $U_0, U_1, \ldots$ with each one being an element of the next one, usually written $U_i : U_{i+1}$ in type theory. However, to keep the system consistent, some care must be taken when defining universe constructors. For instance, if $A : U_i$ and, for all $x : A$, $B(x) : U_j$, then, with predicative universes, we must have $(\Pi x : A, B(x)) : U_{\max(i,j)}$.

Following [14], one can easily encode such an infinite hierarchy of type universes in $\lambda\Pi/\mathcal{R}$, by using the following $\lambda\Pi/\mathcal{R}$ infinite signature and set of rules:[1]

- for each universe $U_i$, the symbols $U_i : \mathrm{Type}$ and $T_i : U_i \to \mathrm{Type}$;
- for each axiom $U_i : U_{i+1}$, the symbol $u_i : U_{i+1}$ and the rewrite rule $T_{i+1} u_i \longrightarrow U_i$;
- for each product from $U_i$ to $U_j$, the symbol $\pi_{i,j} : \Pi x : U_i, (T_i\, x \to U_j) \to U_{\max(i,j)}$ and the rewrite rule $T_{\max(i,j)}(\pi_{i,j}\, x\, y) \longrightarrow \Pi z : T_i\, x, T_j(y\, z)$.

To get a finite signature, one can represent type universes in Peano arithmetic using the following algebra [5]:

▶ **Definition 1** (Max-successor algebra). *The max-successor algebra $\mathcal{L}$ is the first-order term algebra made of the symbols $\mathtt{z}$ of arity 0, $\mathtt{s}$ of arity 1 and $\sqcup$ of arity 2, written infix. We moreover take $\sqcup$ of smaller priority than $\mathtt{s}$ so that $\mathtt{s}x \sqcup y$ is the same as $(\mathtt{s}x) \sqcup y$. Then, let $\mathcal{C} = \mathcal{V} \cup \{\mathtt{z}\}$ where $\mathcal{V}$ some set of variables disjoint from function symbols.*

*The interpretation of a term $t$ wrt a valuation $\mu : \mathcal{V} \to \mathbb{N}$ is as expected:*

- $\mathtt{z}$ *is interpretated as 0:* $\mathtt{z}\mu = 0$,
- $\mathtt{s}$ *is interpreted as the successor function:* $(\mathtt{s}\, t)\mu = t\mu + 1$,
- $\sqcup$ *is interpreted as the binary max function on* $\mathbb{N}$*:* $(u \sqcup v)\mu = \max(u\mu, v\mu)$.

*Two terms $t, u$ are equivalent, written $t \simeq u$, if, for all valuations $\mu$, $t\mu = u\mu$.*

*In the following, we will denote by $\simeq_A$ the equational sub-theory of $\simeq$ generated by the equation $(t \sqcup u) \sqcup v = t \sqcup (u \sqcup v)$, and by $\simeq_{AC}$ the equational sub-theory of $\simeq$ generated by the equations $u \sqcup v = v \sqcup u$ and $(t \sqcup u) \sqcup v = t \sqcup (u \sqcup v)$.*

---

[1] In $\lambda\Pi/\mathcal{R}$, rules are sometimes presented as part of the signature [13, 30].

By using this algebra, we can then encode in $\lambda\Pi/\mathcal{R}$ a type system with an infinite hierarchy of type universes using the following *finite* signature:

- the symbols $\mathcal{L}$ : Type, $\mathtt{z} : \mathcal{L}$, $\mathtt{s} : \mathcal{L} \to \mathcal{L}$, $\sqcup : \mathcal{L} \to \mathcal{L} \to \mathcal{L}$ and the rules $\mathtt{z} \sqcup y \longrightarrow y$, $x \sqcup \mathtt{z} \longrightarrow x$, $(\mathtt{s}\, x) \sqcup (\mathtt{s}\, y) \longrightarrow \mathtt{s}\, (x \sqcup y)$;
- the symbols $U : \mathcal{L} \to \text{Type}$ and $T : \Pi i : \mathcal{L}, U\, i \to \text{Type}$;
- the symbol $u : \Pi i : \mathcal{L}, U\, (\mathtt{s}\, i)$ and the rewrite rule $T\, _\, (u\, i) \longrightarrow U\, i$;
- the symbol $\pi : \Pi i : \mathcal{L}, \Pi j : \mathcal{L}, \Pi x : U\, i, (T\, i\, x \to U\, j) \to U\, (i \sqcup j)$ and the rewrite rule $T\, _\, (\pi\, i\, j\, x\, y) \longrightarrow \Pi z : T\, i\, x, T\, j\, (y\, z)$.

The rules defining $\sqcup$ are indeed sufficient to decide whether $t \simeq u$ when $t$ and $u$ are closed terms (i.e. terms with no variables), which is necessary for deciding the type conversion relation of $\lambda\Pi/\mathcal{R}$.

**Universe variables.** This representation with universe variables is also useful to represent systems with floating/elastic universes or universe polymorphism like in Coq or Agda [33, 32, 2]. However, in this case, the rules defining $\sqcup$ do not allow one to decide $\simeq$ on open terms (i.e. terms with variables), even if one adds the associativity and commutativity of $\sqcup$ in the type conversion because, for instance, $x \sqcup x = x$ ($\sqcup$ is idempotent), $x \sqcup \mathtt{s}x = \mathtt{s}x$, $x \sqcup \mathtt{s}(\mathtt{s}x) = \mathtt{s}(\mathtt{s}x)$, ...

The relation $\simeq$ on open terms is decidable though since it is a sub-theory of Presburger arithmetic [29]. So, one solution could be to use as logical framework not $\lambda\Pi/\mathcal{R}$ but an extension of LF with decision procedures, like CoqMT [8]. But the translation from such a logical framework to HOL-Light, Coq, Agda, etc. would be more difficult or introduce undesirable axioms in the target system.

In [6], Assaf and his coauthors introduced a presentation of the max-successor algebra to deal with universe variables. They replaced the successor symbol $\mathtt{s}$ by two new symbols: $\mathtt{1}$ of arity 0, and $+$ of arity 2. However, they had to use rewriting with matching modulo associativity and commutativity (AC) of $\sqcup$, and associativity, commutativity and unit (ACU) of $+$ (as $\mathtt{z}$ is a neutral element of $+$), and extend type conversion with those theories too. But matching modulo AC or ACU is NP-complete [24, 25].

Finally, in [20], Genestier introduced another presentation of the max-successor algebra that can be decided by using $\simeq_{AC}$ and matching modulo $\simeq_{AC}$ only (more details will be given in Section 3).

However, efficient implementations of matching modulo AC or AC-equivalence rely on data structures for representing terms that are different from the ones used for implementing $\beta$-reduction and type-checking in dependent type systems [4, 35, 12, 1]. For instance, in [15, 16], an AC symbol $f$ is considered as varyadic (i.e. can take any number of arguments) and terms are "flattened" so that $f$ has no argument headed by $f$. The addition of AC-matching and AC-equivalence in a type-checker for $\lambda\Pi/\mathcal{R}$ can therefore introduce inefficiencies and bugs, and greatly increase the size of the code. For instance, the addition of AC-matching and AC-equivalence in Dedukti doubled the size of the code[2].

We can therefore wonder whether there is another way to handle universe variables that is easier to implement in a type-checker for $\lambda\Pi/\mathcal{R}$.

[2] See `https://github.com/Deducteam/Dedukti/pull/219`.

**Outline.** In this paper, we give yet another presentation of the max-successor algebra together with a new convergent rewrite system for deciding it that does not use matching modulo AC. This can be achieved by keeping terms in some AC canonical form, following a technique introduced in [11].

We start by giving a direct proof of decidability of the word problem in the max-successor algebra. This will allow us to introduce some notions, like the one of canonical form, that is at the basis of our new presentation. For the sake of completeness, we then recall Genestier's rewrite system with matching modulo AC. Then, in Section 4, we give a new presentation of the max-successor algebra and a convergent rewrite system for deciding the equivalence of two AC-canonical terms of a shape ensured by our translation. Finally, in Section 5, we explain how to modify the code of a $\lambda\Pi/\mathcal{R}$ type-checker to ensure that every term can always be in AC-canonical form. This work has been implemented in the proof assistant Lambdapi and the code is freely accessible on `https://github.com/Deducteam/lambdapi`.

## 2 Word problem in the max-successor algebra

We first give a direct proof of decidability of $\simeq$ by recalling the notion of canonical form for the max-successor algebra introduced by Genestier in [20], by showing that two equivalent terms have equal canonical forms, and by providing a recursive functional program for computing the canonical form of a term. To this end, we reuse a terminology that is common in the study of hetegeneous signatures [18, 21]:

▶ **Definition 2** (Aliens, combs and caps). *Given a binary symbol $f$, let* $\mathrm{aliens}_f : \mathcal{L} \to \mathcal{L}^+$ *be the function mapping every term to a non-empty list of terms such that* $\mathrm{aliens}_f(t) = \mathrm{aliens}_f(u)\mathrm{aliens}_f(v)$ *(the list concatenation being written by juxtaposition) if $t = fuv$, and* $\mathrm{aliens}_f(t) = t$ *(the singleton list) otherwise.*

*Conversely, let* $\mathrm{comb}_f : \mathcal{L}^+ \to \mathcal{L}$ *be the function mapping a non-empty list of terms to a term such that* $\mathrm{comb}_f[t] = t$ *and, for all $n \geq 2$,* $\mathrm{comb}_f[t_1, \dots, t_n] = ft_1\mathrm{comb}_f[t_2, \dots, t_n]$.

*Let an $f$-context be a term whose symbols are $f$ or a distinguished variable $\square$. Given an $f$-context $C$ with $n$ occurrences of $\square$ at the respective (disjoint) positions*[3] $p_1 < \ldots < p_n$ *(ordered lexicographically*[4]*), and $n$ terms $t_1, \dots, t_n$, let $C[t_1, \dots, t_n]$ be the term obtained by replacing the occurrence of $\square$ at position $p_i$ by $t_i$ for every $i$.*

*Given a term $t$, let* $\mathrm{cap}_f(t)$ *be the (unique) biggest $f$-context $C$ such that* $t = C[\mathrm{aliens}_f(t)]$.

▶ **Example.** $\mathrm{aliens}_\sqcup((x \sqcup y) \sqcup z) = [x; y; z]$, $\mathrm{comb}_\sqcup[x; y; z] = x \sqcup (y \sqcup z)$, $\mathrm{cap}_\sqcup((x \sqcup y) \sqcup z) = ((\square \sqcup \square) \sqcup \square)$, $\mathrm{Pos}((x \sqcup y) \sqcup z) = \{\varepsilon, 1, 2, 11, 12\}$, *and* $\mathrm{cap}_\sqcup((x \sqcup y) \sqcup z)[t_1, t_2, t_3] = (t_1 \sqcup t_2) \sqcup t_3$.

▶ **Lemma 3.**

- *For all terms $t$,* $t \simeq_A \mathrm{comb}_\sqcup(\mathrm{aliens}_\sqcup(t))$.
- *For all sequences of terms $l, m$ and terms $t, u$,* $\mathrm{comb}_\sqcup(ltum) \simeq_{AC} \mathrm{comb}_\sqcup(lutm)$.
- *For all terms $t_1, \dots, t_n$,* $\mathtt{s}(\mathrm{comb}_\sqcup[t_1, \dots, t_n]) \simeq \mathrm{comb}_\sqcup[\mathtt{s}(t_1), \dots, \mathtt{s}(t_n)]$

**Proof.**

- By definition, $t = \mathrm{cap}_\sqcup(t)[\mathrm{aliens}_\sqcup(t)]$. Let $C$ be the canonical form of $\mathrm{cap}_\sqcup(t)$ wrt the convergent rewrite system made of the rewrite rule $(x \sqcup y) \sqcup z \to x \sqcup (y \sqcup z)$. We have $\mathrm{cap}_\sqcup(t) \simeq_A C$, $\mathrm{cap}_\sqcup(t)[\mathrm{aliens}_\sqcup(t)] \simeq_A C[\mathrm{aliens}_\sqcup(t)]$ and $C[\mathrm{aliens}_\sqcup(t)] = \mathrm{comb}_\sqcup(\mathrm{aliens}_\sqcup(t))$. Therefore, $t \simeq_A \mathrm{comb}_\sqcup(\mathrm{aliens}_\sqcup(t))$.

[3] The set $\mathrm{Pos}(t)$ of the positions in a term $t$ is defined as usual as words on $\mathbb{N}$: $\mathrm{Pos}(x) = \{\varepsilon\}$ where $\varepsilon$ is the empty word, and $\mathrm{Pos}(ft_1 \dots t_n) = \{\varepsilon\} \cup \{ip \mid 1 \leq i \leq n, p \in \mathrm{Pos}(t_i)\}$.

[4] $ip < jq$ if $i < j$ or else $i = j$ and $p < q$.

- By induction on $l$.
  - Case $l$ empty. If $m$ is empty, $\mathrm{comb}_\sqcup(tu) \simeq_{AC} \mathrm{comb}_\sqcup(ut)$. Otherwise, $\mathrm{comb}_\sqcup(tum) = t \sqcup (u \sqcup \mathrm{comb}_\sqcup(m)) \simeq_{AC} u \sqcup (t \sqcup \mathrm{comb}_\sqcup(m)) = \mathrm{comb}_\sqcup(utm)$.
  - Case $l = al'$. $\mathrm{comb}_\sqcup(ltum) = a \sqcup \mathrm{comb}_\sqcup(l'tum)$. By induction hypothesis, $\mathrm{comb}_\sqcup(l'tum) \simeq_{AC} \mathrm{comb}_\sqcup(l'utm)$. Therefore, $\mathrm{comb}_\sqcup(ltum) \simeq_{AC} \mathrm{comb}_\sqcup(lutm)$.
- First note that, for all $x$ and $y$, $\mathtt{s}(x \sqcup y) \simeq (\mathtt{s}x) \sqcup (\mathtt{s}y)$. We then proceed by induction on $n$. If $n = 1$, this is immediate since $\mathrm{comb}_\sqcup[t] = t$. If $n \geq 2$, $\mathtt{s}(\mathrm{comb}_\sqcup[t_1, \ldots, t_n]) = \mathtt{s}(t_1 \sqcup \mathrm{comb}_\sqcup[t_2, \ldots, t_n]) \simeq (\mathtt{s}t_1) \sqcup (\mathtt{s}(\mathrm{comb}_\sqcup[t_2, \ldots, t_n]))$. By induction hypothesis, $\mathtt{s}(\mathrm{comb}_\sqcup[t_2, \ldots, t_n]) \simeq \mathrm{comb}_\sqcup[\mathtt{s}(t_2), \ldots, \mathtt{s}(t_n)]$. Therefore, $\mathtt{s}(\mathrm{comb}_\sqcup[t_1, \ldots, t_n]) \simeq \mathrm{comb}_\sqcup[\mathtt{s}(t_1), \ldots, \mathtt{s}(t_n)]$. ◀

▶ **Definition 4** (s-terms, $S$-function and total order on s-terms). *A term is an* s*-term if it contains no* $\sqcup$ *symbol.*

*For all* s*-terms* $t$*, there is a unique pair* $(k, x) \in \mathbb{N} \times \mathcal{C}$ *such that* $t = S\,k\,x$*, where* $S : \mathbb{N} \to \mathcal{L} \to \mathcal{L}$ *is the (meta-level) function such that* $S\,0\,t = t$ *and, for all* $n \geq 1$*,* $S\,n\,t = S\,(n-1)\,(\mathtt{s}\,t)$*.*

*Assuming that* $\mathcal{C}$ *is totally ordered, we define a total order on* s*-terms by taking* $Spx \leq Sqy$ *iff* $x \leq y$ *or else* $x = y$ *and* $p \leq q$*.*

▶ **Definition 5** (Canonical forms). *A term* $t \in \mathcal{L}$ *is in canonical form if:*

- $t = \mathrm{comb}_\sqcup[\mathrm{aliens}_\sqcup(t)]$*,*
- $\mathrm{aliens}_\sqcup(t)$ *is a strictly increasing list of* s*-terms (in the order of Definition 4),*
- $t$ *is linear (every variable occurs at most once),*
- *if* $S\,k\,\mathtt{z}$ *and* $S\,l\,x$ *are aliens of* $t$ *then* $k > l$*.*

▶ **Lemma 6.**

- *Two equivalent canonical forms are equal.*
- *Every term is equivalent to a canonical form.*

**Proof.**

- Let $t$ and $u$ be two equivalent canonical forms. $t$ and $u$ have the same variables $x_1, \ldots, x_n$ since, otherwise, they could not have the same interpretation for all valuations. Let $Sk_1x_1, \ldots, Sk_nx_n$ be the aliens of $t$ not of the form $Sk\mathtt{z}$, and $Sl_1x_1, \ldots, Sl_nx_n$ be the aliens of $u$ not of the form $Sk\mathtt{z}$.
  Assume that $t$ has an alien of the form $Sk_0\mathtt{z}$ and $u$ has no alien of the form $Sk\mathtt{z}$. Then, $n > 0$ and $0 \leq k_n < k_0$. But, by taking $x_i\mu = 0$ for all $i$, we get $t\mu = k_0$ and $u\mu = 0$, which is not possible since $t \simeq u$.
  Assume that $t$ has an alien of the form $Sk_0\mathtt{z}$ and $u$ has an alien of the form $Sl_0\mathtt{z}$. By taking $x_i\mu = 0$ for all $i$, we get $t\mu = k_0$ and $u\mu = l_0$. Therefore, $k_0 = l_0$.
  Let now $M = \max(\{k_i | 1 \leq i \leq n\} \cup \{l_i | 1 \leq i \leq n\})$, $N = \max(M, k)$ if $t$ and $u$ have an alien of the form $Sk\mathtt{z}$, and $N = M$ otherwise. For all $i \geq 1$, let $\mu_i$ be the valuation mapping $x_i$ to $N$ and all other variables to 0. Then, $t\mu = N + k_i$ and $u\mu = N + l_i$. Therefore, $k_i = l_i$ for all $i$, and $t = u$.
- We prove that, for all terms $t$, there is a canonical form $t'$ such that $t \simeq t'$, by induction on the size of $t$.
  - Case $t$ is a variable or $\mathtt{z}$. This is immediate since $t$ is in canonical form.
  - Case $t = \mathtt{s}u$. By induction hypothesis, $u \simeq u'$ in canonical form. Let $[u_1, \ldots, u_n]$ be the aliens of $u'$. We have $t \simeq \mathtt{s}u' = \mathtt{s}(\mathrm{comb}_\sqcup[u_1, \ldots, u_n]) \simeq \mathrm{comb}_\sqcup[\mathtt{s}(u_1), \ldots, \mathtt{s}(u_n)]$. $[\mathtt{s}(u_1), \ldots, \mathtt{s}(u_n)]$ is a strictly increasing list of s-terms. Moreover, if $Sk\mathtt{z}$ and $Slx$ are elements of this list, then $k > l$. Therefore, $\mathrm{comb}_\sqcup[\mathtt{s}(u_1), \ldots, \mathtt{s}(u_n)]$ is a canonical form.

- Case $t = u \sqcup v$. By induction hypothesis, $u \simeq u'$ in canonical form, and $v \simeq v'$ in canonical form. Given a list of $\mathtt{s}$-terms, let $\mathrm{sort}(l)$ be the function putting the elements of $l$ in increasing order. We have $\mathrm{comb}_\sqcup(l) \simeq_{AC} \mathrm{comb}_\sqcup(\mathrm{sort}(l))$. Given an increasing list of $\mathtt{s}$-terms, let $\mathrm{merge}(l)$ be the function that, starting from $l$:
  * replaces any two (adjacent) terms $Spx$, $Sqx$ by the single term $S(p \sqcup_{\mathbb{N}} q)x$,
  * removes any term $Sp\mathtt{z}$ if there is also some term $Sqx$ with $p \leq q$.

  We have $\mathrm{comb}_\sqcup(l) \simeq \mathrm{comb}_\sqcup(\mathrm{merge}(l))$ since $Spx \sqcup Sqx \simeq S(p \sqcup_{\mathbb{N}} q)x$ and $Sp\mathtt{z} \sqcup Sqx \simeq Sqx$ if $p \leq q$. Let now $l = \mathrm{aliens}_\sqcup(u')$ and $m = \mathrm{aliens}_\sqcup(v')$. Then, $t \simeq u' \sqcup v' = \mathrm{comb}_\sqcup(\mathrm{aliens}_\sqcup(u' \sqcup v')) = \mathrm{comb}_\sqcup(lm) \simeq_{AC} \mathrm{comb}_\sqcup(\mathrm{sort}(lm)) \simeq \mathrm{comb}_\sqcup(\mathrm{merge}(\mathrm{sort}(lm)))$, which is in canonical form. ◀

It follows that, for checking whether $t \simeq u$, it suffices to compute and syntactically compare the canonical forms of $t$ and $u$. This could be easily done in any programming language. However, we are interested in implementing this in the logical framework $\lambda\Pi/\mathcal{R}$ and its implementation Lambdapi, which allows one to define functions by using rewriting rules with syntactic matching only. However, before showing that this can indeed be done, we are first going to see a solution using rewriting with matching modulo AC proposed in [20] and implemented in Dedukti thanks to the addition of matching modulo AC in Dedukti by Gaspard Férey (see p. 92 in [17]).

## 3 Decision procedure using matching modulo AC

In this section, we recall the rewriting system using matching modulo AC proposed by Genestier in [20] for deciding $\simeq$. The idea is to represent the terms of $\mathcal{L}$ as the maximum of a natural number and of a finite set of expressions corresponding to the terms $S\,l\,x$ with $x$ a variable. To do so, Genestier uses a multi-sorted term algebra with three sorts:[5]

- The sort $\mathtt{N}$ with the constructors $0 : \mathtt{N}$, $\mathtt{s} : \mathtt{N} \to \mathtt{N}$, $+ : \mathtt{N} \times \mathtt{N} \to \mathtt{N}$ and $\oplus : \mathtt{N} \times \mathtt{N} \to \mathtt{N}$ written infix, with $\oplus$ of priority smaller than $\mathtt{s}$, to represent arithmetic expressions on natural numbers. The sort $\mathtt{N}$ is interpreted as $\mathbb{N}$, $0$ as $0$, $\mathtt{s}$ as the successor function, $+$ as the addition, and $\oplus$ as the maximum.
- The sort $\mathtt{E}$ with the constructors $\emptyset : \mathtt{E}$, $\mathtt{a} : \mathtt{N} \times \mathtt{L} \to \mathtt{E}$, $\cup : \mathtt{E} \times \mathtt{E} \to \mathtt{E}$ written infix, and $\mathtt{A} : \mathtt{N} \times \mathtt{E} \to \mathtt{E}$, to represent the maximum of a finite set of arithmetic expressions. The sort $\mathtt{E}$ is interpreted as $\mathbb{N} \cup \{-\infty\}$, $\emptyset$ as $-\infty$, $\mathtt{a}$ and $\mathtt{A}$ as the addition with $x + (-\infty) = -\infty$, and $\cup$ as the maximum. $\mathtt{a}\,k\,t$ represents the singleton set $\{k + t\}$, and the auxiliary function $\mathtt{A}\,k\,E$ (called `mapPlus` in [20]) adds $k$ to every element of $E$.
- The sort $\mathtt{L}$ with the constructor $\mathtt{m} : \mathtt{N} \times \mathtt{E} \to \mathtt{L}$. The sorts $\mathtt{L}$ is interpreted as $\mathbb{N}$, and $\mathtt{m}$ as the maximum.

A term of $\mathcal{L}$ is translated to a term of sort $\mathtt{L}$ with the same interpretation as follows:

- $|x| = x$,
- $|\mathtt{z}| = \mathtt{m}\,0\,\emptyset$,
- $|\mathtt{s}\,t| = \mathtt{m}\,(\mathtt{s}\,0)\,(\mathtt{a}\,(\mathtt{s}\,0)\,|t|)$
- $|u \sqcup v| = \mathtt{m}\,0\,((\mathtt{a}\,0\,|u|) \cup (\mathtt{a}\,0\,|v|))$

Then, Genestier introduces a rewrite system, that we will call $\mathcal{G}$, made of the rewrite rules of Figure 1 and of the rewrite rules of Figure 2. The second rule for $\cup$ corresponds to the equation $(p + x) \oplus (q + x) = (p \oplus q) + x$. It allows one to have at most one occurrence of

[5] In [20], $\sqcup$ is denoted by `max`, $\mathtt{E}$ by `LSet`, $\mathtt{a}$ by $\oplus$, $\mathtt{A}$ by `mapPlus`, and $\mathtt{m}$ by `Max`.

$$
\begin{array}{rcl}
0 + q & \longrightarrow & q \\
\mathtt{s}\,p + q & \longrightarrow & \mathtt{s}\,(p + q) \\
p \oplus 0 & \longrightarrow & p \\
0 \oplus q & \longrightarrow & q \\
\mathtt{s}\,p \oplus \mathtt{s}\,q & \longrightarrow & \mathtt{s}\,(p \oplus q)
\end{array}
$$

**Figure 1** Rewrite rules for addition and maximum on natural numbers.

$$
\begin{array}{rcl}
X \cup \emptyset & \longrightarrow & X \\
(\mathtt{a}\,p\,x) \cup (\mathtt{a}\,q\,x) & \longrightarrow & \mathtt{a}\,(p \oplus q)\,x \\
\mathtt{A}\,p\,\emptyset & \longrightarrow & \emptyset \\
\mathtt{A}\,p\,(\mathtt{a}\,q\,x) & \longrightarrow & \mathtt{a}\,(p + q)\,x \\
\mathtt{A}\,p\,(X \cup Y) & \longrightarrow & (\mathtt{A}\,p\,X) \cup (\mathtt{A}\,p\,Y) \\
\mathtt{m}\,0\,(\mathtt{a}\,0\,x) & \longrightarrow & x \\
\mathtt{m}\,p\,(\mathtt{a}\,q\,(\mathtt{m}\,r\,X)) & \longrightarrow & \mathtt{m}\,(p \oplus (q + r))\,(\mathtt{A}\,q\,X) \\
\mathtt{m}\,p\,((\mathtt{a}\,q\,(\mathtt{m}\,r\,X)) \cup Y) & \longrightarrow & \mathtt{m}\,(p \oplus (q + r))\,((\mathtt{A}\,q\,X) \cup Y)
\end{array}
$$

**Figure 2** The system $\mathcal{G}$ for computing canonical forms with matching modulo AC includes the above rules as well as the rules of Figure 1.

every variable $x$. The second rule of $\mathtt{A}$ corresponds to the equation $p + (q + x) = (p + q) + x$, while the third rule of $\mathtt{A}$ corresponds to the equation $p + (x \oplus y) = (p + x) \oplus (p + y)$. The rules of $\mathtt{m}$ are the main rules for computing the canonical form. The first rule corresponds to the equation $0 \oplus (0 + x) = x$. The second rule corresponds to the equation $p \oplus (q + (r \oplus (k_1 + x_1) \oplus \ldots \oplus (k_n + x_n))) = (p \oplus (q + r)) \oplus (q + k_1 + x_1) \oplus \ldots (q + k_n + x_n)$. The last rule is similar.

Genestier then proves the following properties:

- The rewrite relation $\longrightarrow_{\mathcal{G},AC}$ generated by $\mathcal{G}$ using matching modulo associativity and commutativity of $\cup$ is not confluent on terms with variables of sort $\mathtt{N}$ or $\mathtt{E}$.
- For all terms $t$ in $\mathcal{L}$, any $\longrightarrow_{\mathcal{G},AC}$-normal form of $|t|$ is either a variable or of the form $\mathtt{m}\,p\,((\mathtt{a}\,q_1\,x_1) \cup \ldots \cup (\mathtt{a}\,q_n\,x_n))$ with $x_1, \ldots, x_n$ distinct variables and, for all $k$, $q_k \leq p$.
- Two such normal forms are equal modulo associativity-commutativity of $\cup$.

To these results, we can add:

▶ **Lemma 7.** *The relation $\longrightarrow_{\mathcal{G}/AC} = \simeq_{AC}\longrightarrow_{\mathcal{G}}\simeq_{AC}$ generated by $\mathcal{G}$ on AC-equivalence classes, which contains $\longrightarrow_{\mathcal{G},AC}$, terminates.*

**Proof.** It can be automatically proved by, for instance AProVE [3], using 3 consecutive strictly monotone polynomial interpretations on $\mathbb{N}$, and then formally certified in Isabelle/HOL by CeTA[6]. ◀

[6] http://cl-informatik.uibk.ac.at/software/ceta/

## 4 Getting rid of matching modulo AC

In this section, we present our main contribution: a new presentation of $\mathcal{L}$ and a new rewrite system not using matching modulo AC. It is inspired by the decidability proof of Section 2.

The main problem for computing the canonical form of a term is to be able to replace an expression of the form $Spx \sqcup (Sry \sqcup Sqx)$ by $S(p \oplus q)x \sqcup Sry$. One way to do it is by using the rule (4) of Figure 3 with matching modulo AC. Indeed, we have $Spx \sqcup (Sry \sqcup Sqx) \simeq_{AC} Spx \sqcup (Sqx \sqcup Sry) \longrightarrow_{\mathcal{R}} S(p \oplus q)x \sqcup Sry$. Another way to do it is to make sure that the aliens of a term are always ordered so that two aliens $Spx$ and $Sqx$ sharing the same variable $x$ are always put side by side. Following [11], this can be achieved by replacing constructors by construction functions, that is here, $\sqcup$ by some new function symbol $\sqcup'$ which will rearrange its aliens so as to get such an AC-canonical form. Hence, we get $Spx \sqcup' (Sry \sqcup' Sqx) \simeq_{AC} Spx \sqcup (Sqx \sqcup Sry) \longrightarrow_{\mathcal{R}} S(p \oplus q)x \sqcup Sry$.

Again, we translate terms of $\mathcal{L}$ into a multi-sorted term algebra. However, our algebra is simpler than Genestier's algebra. Like [20], we distinguish expressions representing natural numbers from the other expressions by using distinct sorts. However, we do not introduce a new sort for sets but simply extend $\mathcal{L}$-terms with a new symbol $\mathtt{S}$ corresponding to the (meta-level) function $S$ of Definition 4.

We consider the multi-sorted term algebra $\mathcal{I}$ with two sorts $\mathtt{N}$ and $\mathtt{L}$, and the constructors $0 : \mathtt{N}$, $\mathtt{s} : \mathtt{N} \to \mathtt{N}$, $+ : \mathtt{N} \times \mathtt{N} \to \mathtt{N}$ and $\oplus : \mathtt{N} \times \mathtt{N} \to \mathtt{N}$ written infix, $\mathtt{z} : \mathtt{L}$, $\mathtt{S} : \mathtt{N} \times \mathtt{L} \to \mathtt{L}$ and $\sqcup : \mathtt{L} \to \mathtt{L} \to \mathtt{L}$. Again, we assume that $\oplus$ is of priority smaller than $\mathtt{s}$. All the sorts are interpreted as $\mathbb{N}$, 0 as 0, $\mathtt{s}$ as the successor function, $+$ and $\mathtt{S}$ as the addition, and $\oplus$ as the maximum.

▶ **Definition 8** (Guarded terms). *An $\mathcal{I}$-term is guarded if every occurrence of an element $x \in \mathcal{C}$ of sort $\mathtt{L}$ is in a subterm of the form $\mathtt{S}\,p\,x$.*

The idea behind guarded terms is to represent an $\mathcal{L}$-term of the form $S\,k\,x$ by the $\mathcal{I}$-term $\mathtt{S}\,\overline{k}\,x$, where $\overline{k}$ is the representation of $k$ in $\mathtt{N}$.

An $\mathcal{L}$-term is translated into a guarded $\mathcal{I}$-term of sort $\mathtt{L}$ with the same interpretation in $\mathbb{N}$ as follows:

- $|x| = \mathtt{S}\,0\,x \sqcup \mathtt{S}\,0\,\mathtt{z}$
- $|\mathtt{z}| = \mathtt{S}\,0\,\mathtt{z}$
- $|\mathtt{s}\,t| = \mathtt{S}\,(\mathtt{s}\,0)\,|t|$
- $|u \sqcup v| = |u| \sqcup |v|$

For each occurrence of a variable, we add an occurrence of $\mathtt{z}$ so that, after normalization (see below), we get a term of the form $\mathtt{S}\,p_1\,x_1 \sqcup \ldots \sqcup \mathtt{S}\,p_n\,x_n \sqcup \mathtt{S}\,q\,\mathtt{z}$ with $p_i \leq q$.

▶ **Definition 9** (AC-canonical forms). *Let $\leq$ be any total order on $\mathcal{I}$-terms such that $\mathtt{S}\,p\,x \leq \mathtt{S}\,q\,y$ iff $x < y$ or else $x = y$ and $p \leq q$.*[7]

*An $\mathcal{I}$-term $t$ is in AC-canonical form if $t = \mathrm{comb}_\sqcup[\mathrm{sort}(\mathrm{aliens}_\sqcup(t))]$ and every element of $\mathrm{aliens}_\sqcup(t) - \{t\}$ is in AC-canonical form, where $\mathrm{sort}(l)$ is the elements of $l$ in increasing order wrt $\leq$.*

*Let $\twoheadrightarrow^{AC}$ be the relation mapping every term $t$ to its unique AC-canonical form $[t]$.*

Two terms are AC-equivalent iff their AC-canonical forms are equal.

---

[7] Take for instance the lexicographic path ordering generated by any total precedence on function symbols and variables, and right-to-left comparison of the arguments of $\mathtt{S}$.

Note that AC-canonization is a canonizer in the sense of Shostak [31]. It satisfies the properties (CAN-1) to (CAN-5) explicited in [26]: (CAN-1) it is idempotent; (CAN-2) it decides $\simeq_{AC}$; (CAN-3) it preserves variables; (CAN-4) every subterm of a canonical term is canonical; and (CAN-5) it commutes with order-preserving variable renamings.

We now introduce the rewrite relation that we will use to decide $\simeq$:

▶ **Definition 10** (Rewriting modulo AC-canonization). *Let $\longrightarrow_{\mathcal{R}}^{AC} = \longrightarrow_{\mathcal{R}}\twoheadrightarrow^{AC}$, where $\mathcal{R}$ is made of the rewrite rules of Figures 1 and 3.*

An $\longrightarrow_{\mathcal{R}}^{AC}$ step is a standard $\longrightarrow_{\mathcal{R}}$ step with syntactic matching followed by AC-canonization. We will see in Section 5 that AC-canonization is easily implemented by replacing constructors by construction functions, so that AC-canonization is implicitly done at term construction time [11]. In other words, our decision procedure reduces to standard rewriting with syntactic matching but on a restricted set of terms, namely the terms in AC-canonical form.

This notion of rewriting is close to the notion of normal rewriting [27], which consists in applying a standard rewrite step after normalization wrt a convergent rewrite system $\mathcal{S}$. The difference is that AC-canonization cannot defined by a convergent rewrite system.

One can easily check that the rules of $\mathcal{R}$ preserve guardedness (if $t$ is guarded and $t \longrightarrow_{\mathcal{R}}^{AC} u$, then $u$ is guarded too) and are semantically correct ($\longrightarrow_{\mathcal{R}}^{AC} \subseteq \simeq$). Indeed, the first rule corresponds to the associativity of $+$: $p+(q+x) = (p+q)+x$. The second rule corresponds to the distributivity of $+$ over $\oplus$: $p+(x\oplus y) = (p+x)\oplus(p+y)$. On the contrary, the last two rules factorize identical monoms that are side by side: $(p+x)\oplus(q+x) = (p\oplus q)+x$.

$$
\begin{array}{lrcl}
(1) & \mathtt{S}\,p\,(\mathtt{S}\,q\,x) & \longrightarrow & \mathtt{S}\,(p+q)\,x \\
(2) & \mathtt{S}\,p\,(x \sqcup y) & \longrightarrow & \mathtt{S}\,p\,x \sqcup \mathtt{S}\,p\,y \\
(3) & \mathtt{S}\,p\,x \sqcup \mathtt{S}\,q\,x & \longrightarrow & \mathtt{S}\,(p \oplus q)\,x \\
(4) & \mathtt{S}\,p\,x \sqcup (\mathtt{S}\,q\,x \sqcup y) & \longrightarrow & \mathtt{S}\,(p \oplus q)\,x \sqcup y
\end{array}
$$

**Figure 3** Rewrite system on canonical forms.

We now prove that the relation $\longrightarrow_{\mathcal{R}}^{AC}$ terminates and is confluent on guarded terms with no variables of sort $\mathtt{N}$.

▶ **Lemma 11.** *The relation $\longrightarrow_{\mathcal{R}/AC} = \simeq_{AC}\longrightarrow_{\mathcal{R}}\simeq_{AC}$, which contains $\longrightarrow_{\mathcal{R}}^{AC}$, terminates.*

**Proof.** AProVE[8] automatically proves the termination of $\longrightarrow_{\mathcal{R}/AC}$ by a succession of 3 strictly monotone polynomial interpretations on $\mathbb{N}$, and its result can be formally checked by CeTA:

- $P_{\mathtt{S}}x_1x_2 = 3 + x_1 + 3x_1x_2 + 3x_2$
- $P_{+}x_1x_2 = x_1 + 2x_1x_2 + x_2$
- $P_{\sqcup}x_1x_2 = 3 + x_1 + x_2$
- $P_{\mathtt{s}}x_1 = x_1$
- $P_{\oplus}x_1x_2 = 1 + x_1 + x_2$
- $P_0 = 1$

validates all the rules as well as the AC axioms of $\sqcup$[9] and strictly orients all the rules except the last rules of $+$ and $\oplus$.

[8] `http://aprove.informatik.rwth-aachen.de/`
[9] A polynomial $Pxy$ validates the AC axioms iff $Pxy = axy + b(x+y) + c$ with $b(b-1) = ac$.

- $P_+ x_1 x_2 = x_1 + x_2$
- $P_\sqcup x_1 x_2 = 3 + 3x_1 + 2x_1x_2 + 3x_2$
- $P_{\mathtt{s}} x_1 = 3 + x_1$
- $P_\oplus x_1 x_2 = 1 + x_1 + 2x_2$

validates all the rules and equations and strictly orients the last rule of $\oplus$.

- $P_+ x_1 x_2 = 3 + 3x_1 + 2x_1x_2 + 2x_2$
- $P_\sqcup x_1 x_2 = 3 + 3x_1 + 2x_1x_2 + 3x_2$
- $P_{\mathtt{s}} x_1 = 3 + 2x_1$

validates all the rules and equations and strictly orients the last rule of $+$. ◀

▶ **Lemma 12.** *The rewrite relation $\longrightarrow_{\mathcal{N}}$ generated by the rules of Figure 1 terminates and is confluent. Moreover, for all closed terms $p, q, r$ of sort* N*, the following pairs of terms are joinable with $\longrightarrow_{\mathcal{N}}$:*

- $(p + q) + r = p + (q + r)$
- $p + q = q + p$
- $(p \oplus q) \oplus r = p \oplus (q \oplus r)$
- $p \oplus q = q \oplus p$
- $p + (q \oplus r) = (p + q) \oplus (p + r)$

**Proof.** The relation $\longrightarrow_{\mathcal{N}}$ terminates since it is included in the lexicographic path ordering with $+, \oplus > \mathtt{s}$. It is confluent since it is weakly orthogonal. So, every term of sort N has a unique normal form. Hence, it is sufficient to prove that the above equations are valid in the equational theory generated by $\mathcal{N}$.

A closed term of sort N in normal form wrt $\longrightarrow_{\mathcal{N}}$ cannot contain a subterm of the form $p + q$ or $p \oplus q$ since, otherwise, the smallest such subterm would be reducible by one of the rules of $\mathcal{N}$. Hence, every closed term of sort N in normal form wrt $\longrightarrow_{\mathcal{N}}$ is of the form $Sk0$ with $k \in \mathbb{N}$, where the (meta-level) function $S$ is defined in Definition 4.

It therefore suffices to prove the above equations by using only induction on natural numbers and the rules of $\mathcal{N}$. This can easily be done in Lambdapi for instance. See `https://github.com/fblanqui/lib`. ◀

▶ **Lemma 13.** *$\longrightarrow_{\mathcal{R}}^{AC}$ is locally confluent on AC-canonical guarded terms with no variables of sort* N*.*

**Proof.** We show that every critical pair is joinable using $\longrightarrow_{\mathcal{R}}^{AC}$ and Lemma 12. In the following, the terms that are not between square brackets are in AC-canonical form. We also write $[p \oplus q]$ to denote either $p \oplus q$ or $q \oplus p$.

**(1)** $\mathtt{S}\,p\,(\mathtt{S}\,q\,x) \longrightarrow \mathtt{S}\,(p+q)\,x$ is overlapped by:
  (1) By taking $x = \mathtt{S}rx$. We have
  $t = \mathtt{S}p(\mathtt{S}q(\mathtt{S}rx)) \longrightarrow_1^{AC} \mathtt{S}(p+q)(\mathtt{S}rx) \longrightarrow_1^{AC} \mathtt{S}((p+q)+r)x$
  and $t \longrightarrow_1^{AC} \mathtt{S}p(\mathtt{S}(q+r)x) \longrightarrow_1^{AC} \mathtt{S}(p+(q+r))x$.
  (2) By taking $x = x \sqcup y$. We have
  $t = \mathtt{S}p(\mathtt{S}q(x \sqcup y)) \longrightarrow_1^{AC} \mathtt{S}(p+q)(x \sqcup y) \longrightarrow_2^{AC} \mathtt{s}(p+q)x \sqcup \mathtt{S}(p+q)y$
  and $t \longrightarrow_2^{AC} \mathtt{S}p(\mathtt{S}qx \sqcup \mathtt{S}qy) \longrightarrow_2^{AC} \mathtt{S}p(\mathtt{S}qx) \sqcup \mathtt{S}p(\mathtt{S}qy)$
  $\longrightarrow_1^{AC} [\mathtt{S}(p+q)x \sqcup \mathtt{S}p(\mathtt{S}qy)] \longrightarrow_1^{AC} \mathtt{S}(p+q)x \sqcup \mathtt{S}(p+q)y$.

**(2)** $\mathtt{S}\,p\,(x \sqcup y) \longrightarrow \mathtt{S}\,p\,x \sqcup \mathtt{S}\,p\,y$ is overlapped by:
  (3) By taking $x = \mathtt{S}qx$ and $y = \mathtt{S}rx$. We have
  $t = \mathtt{S}p(\mathtt{S}qx \sqcup \mathtt{S}rx) \longrightarrow_2^{AC} \mathtt{S}p(\mathtt{S}qx) \sqcup \mathtt{S}p(\mathtt{S}rx) \longrightarrow_1^{AC} [\mathtt{S}(p+q)x \sqcup \mathtt{S}p(\mathtt{S}rx)]$
  $\longrightarrow_1^{AC} [\mathtt{S}(p+q)x \sqcup \mathtt{S}(p+r)x] \longrightarrow_3^{AC} \mathtt{S}[(p+q) \oplus (p+r)]$
  and $t \longrightarrow_3^{AC} \mathtt{S}p(\mathtt{S}(q \oplus r)x) \longrightarrow_1^{AC} \mathtt{S}(p+(q \oplus r))x$.

(4) By taking $x = \mathtt{S}qx$ and $y = \mathtt{S}rx \sqcup y$. We have
$t = \mathtt{S}p(\mathtt{S}qx \sqcup (\mathtt{S}rx \sqcup y)) \longrightarrow_2^{AC} [\mathtt{S}p(\mathtt{S}qx) \sqcup \mathtt{S}p(\mathtt{S}rx \sqcup y)]$
$\longrightarrow_1^{AC} [\mathtt{S}(p+q)x \sqcup \mathtt{S}p(\mathtt{S}rx \sqcup y)] \longrightarrow_2^{AC} [\mathtt{S}(p+q)x \sqcup (\mathtt{S}p(\mathtt{S}rx) \sqcup \mathtt{S}py)]$
$\longrightarrow_1^{AC} [\mathtt{S}(p+q)x \sqcup (\mathtt{S}(p+r)x \sqcup \mathtt{S}py)] \longrightarrow_4^{AC} [\mathtt{S}[(p+q) \oplus (p+r)]x \sqcup \mathtt{S}py]$
and $t \longrightarrow_4^{AC} [\mathtt{S}p(\mathtt{S}(q \oplus r)x \sqcup y)] \longrightarrow_2^{AC} [\mathtt{S}p(\mathtt{S}(q \oplus r)x) \sqcup \mathtt{S}py]$
$\longrightarrow_1^{AC} [\mathtt{S}(p + (q \oplus r))x \sqcup \mathtt{S}py]$.

**(3)** $\mathtt{S}\,p\,x \sqcup \mathtt{S}\,q\,x \longrightarrow \mathtt{S}\,(p \oplus q)\,x$ is overlapped by:

(1) By taking $x = \mathtt{S}rx$. We have
$t = \mathtt{S}p(\mathtt{S}rx) \sqcup \mathtt{S}q(\mathtt{S}rx) \longrightarrow_3^{AC} \mathtt{S}(p \oplus q)(\mathtt{S}rx) \longrightarrow_1^{AC} \mathtt{S}((p \oplus q) + r)x$
and $t \longrightarrow_1^{AC} [\mathtt{S}(p+r)x \sqcup \mathtt{S}q(\mathtt{S}rx)] \longrightarrow_1^{AC} [\mathtt{S}(p+r)x \sqcup \mathtt{S}(q+r)x]$
$\longrightarrow_3^{AC} \mathtt{S}[(p+r) \oplus (q+r)]x$.

(2) By taking $x = x \sqcup y$. We have
$t = \mathtt{S}p(x \sqcup y) \sqcup \mathtt{S}q(x \sqcup y) \longrightarrow_3^{AC} \mathtt{S}(p \oplus q)(x \sqcup y) \longrightarrow_2^{AC} [\mathtt{S}(p \oplus q)x \sqcup \mathtt{S}(p \oplus q)y]$
and $t \longrightarrow_2^{AC} [(\mathtt{S}px \sqcup \mathtt{S}py) \sqcup \mathtt{S}q(x \sqcup y)] \longrightarrow_2^{AC} [(\mathtt{S}px \sqcup \mathtt{S}py) \sqcup (\mathtt{S}qx \sqcup \mathtt{S}qy)]$
$\longrightarrow_3^{AC} [\mathtt{S}px \sqcup (\mathtt{S}qx \sqcup \mathtt{S}(p \oplus q)y)] \longrightarrow_4^{AC} [\mathtt{S}(p \oplus q)x \sqcup \mathtt{S}(p \oplus q)y]$.

**(4)** $\mathtt{S}\,p\,x \sqcup (\mathtt{S}\,q\,x \sqcup y) \longrightarrow \mathtt{S}\,(p \oplus q)\,x \sqcup y$ is overlapped by:

(1) By taking $x = \mathtt{S}rx$. We have
$t = \mathtt{S}p(\mathtt{S}rx) \sqcup (\mathtt{S}q(\mathtt{S}rx) \sqcup y) \longrightarrow_4^{AC} [\mathtt{S}(p \oplus q)(\mathtt{S}rx) \sqcup y]$
$\longrightarrow_1^{AC} [\mathtt{S}((p \oplus q) + r)x \sqcup y]$
and $t \longrightarrow_1^{AC} [\mathtt{S}(p+r)x \sqcup (\mathtt{S}q(\mathtt{S}rx) \sqcup y)]$
$\longrightarrow_1^{AC} [\mathtt{S}(p+r)x \sqcup (\mathtt{S}(q+r)x \sqcup y)] \longrightarrow_4^{AC} [\mathtt{S}[(p+r) \oplus (q+r)]x \sqcup y]$.

(2) By taking $x = x_1 \sqcup x_2$. We have
$t = \mathtt{S}p(x_1 \sqcup x_2) \sqcup (\mathtt{S}q(x_1 \sqcup x_2) \sqcup y) \longrightarrow_4^{AC} [\mathtt{S}(p \oplus q)(x_1 \sqcup x_2) \sqcup y]$
$\longrightarrow_2^{AC} [\mathtt{S}(p \oplus q)x_1 \sqcup (\mathtt{S}(p \oplus q)x_2 \sqcup y)] = u$
and $t \longrightarrow_2^{AC} [(\mathtt{S}px_1 \sqcup \mathtt{S}px_2) \sqcup (\mathtt{S}q(x_1 \sqcup x_2) \sqcup y)]$
$\longrightarrow_2^{AC} [(\mathtt{S}px_1 \sqcup \mathtt{S}px_2) \sqcup ((\mathtt{S}qx_1 \sqcup \mathtt{S}qx_2) \sqcup y)] = v$.
Since $t$ is guarded, wlog we can assume that
$\mathrm{aliens}_\sqcup(y) = l_1, \mathtt{S}r_1x_1, .., \mathtt{S}r_mx_1, l_2, \mathtt{S}s_1x_2, .., \mathtt{S}s_nx_2, l_3$.
Then, $u$ can be reduced to $\mathrm{comb}_\sqcup[l_1, \mathtt{S}ax_1, l_2, \mathtt{S}bx_2, l_3]$, where
$a = \mathrm{comb}_\oplus[r_1, .., p \oplus q, .., r_m]$ and $b = \mathrm{comb}_\oplus[s_1, .., p \oplus q, .., s_n]$,
by applying $m + n$ times $\longrightarrow_4^{AC}$,
and $v$ can be reduced to $\mathrm{comb}_\sqcup[l_1, \mathtt{S}a'x_1, l_2, \mathtt{S}b'x_2, l_3]$, where
$a' = \mathrm{comb}_\oplus[r_1, .., p, .., q, .., r_m]$ and $b' = \mathrm{comb}_\oplus[s_1, .., p, .., q, .., s_n]$,
by applying $m + n + 2$ times $\longrightarrow_4^{AC}$.

(3) By taking $y = \mathtt{S}rx$. We have
$t = \mathtt{S}px \sqcup (\mathtt{S}qx \sqcup \mathtt{S}rx) \longrightarrow_4^{AC} [\mathtt{S}(p \oplus q)x \sqcup \mathtt{S}rx] \longrightarrow_3^{AC} \mathtt{S}((p \oplus q) \oplus r)x$
and $t \longrightarrow_3^{AC} [\mathtt{S}px \sqcup \mathtt{S}(q \oplus r)x] \longrightarrow_3^{AC} \mathtt{S}(p \oplus (q \oplus r))x$.

(4) By taking $y = \mathtt{S}rx \sqcup y$. We have
$t = \mathtt{S}px \sqcup (\mathtt{S}qx \sqcup (\mathtt{S}rx \sqcup y)) \longrightarrow_4^{AC} [\mathtt{S}(p \oplus q)x \sqcup (\mathtt{S}rx \sqcup y)] = u$
and $t \longrightarrow_4^{AC} [\mathtt{S}px \sqcup (\mathtt{S}(q \oplus r)x \sqcup y)] = v$.
Since $t$ is guarded, wlog we can assume that $\mathrm{aliens}_\sqcup(y) = \mathtt{S}r_1x, .., \mathtt{S}r_mx, l$.
Then, $u$ can be reduced to $\mathrm{comb}_\sqcup[\mathtt{S}ra, l]$, where
$a = \mathrm{comb}_\oplus[r_0, .., p \oplus q, .., r_m]$ and $r_0 = r$, by applying $m + 1$ times $\longrightarrow_4^{AC}$,
and $v$ can be reduced to $\mathrm{comb}_\sqcup[\mathtt{S}a'x, l]$,
where $a' = \mathrm{comb}_\oplus[r_0, .., p, .., q, .., r_m]$, by applying $m + 2$ times $\longrightarrow_4^{AC}$. ◀

Hence, every $\mathcal{L}$-term has, after translation into an $\mathcal{I}$-term, a unique normal form wrt $\longrightarrow_\mathcal{R}^{AC}$. We now prove that this normal form is almost a canonical form, and that it is sufficient to decide $\simeq$.

▶ **Lemma 14.** *For all $\mathcal{L}$-terms $t$ and $u$, we have $t \simeq u$ iff $[|t|]$ and $[|u|]$ have the same normal form wrt $\longrightarrow_{\mathcal{R}}^{AC}$, where $[|t|]$ is the AC-canonical form of the translation of $t$ in $\mathcal{I}$.*

**Proof.** Wlog we can assume that $x \leq \mathtt{z}$ for all $x$.

Let $\mathcal{T}$ be the set of $\mathcal{I}$-terms containing $\mathtt{z}$ that are guarded and have no variable of sort $\mathtt{N}$.

First note that every $\mathcal{T}$-term that is in normal form wrt $\longrightarrow_{\mathcal{R}}^{AC}$ is of the form $\mathtt{S}\,p_1\,x_1 \sqcup \ldots \sqcup \mathtt{S}\,p_n\,x_n \sqcup \mathtt{S}\,q\,\mathtt{z}$ with $x_1 < \ldots < x_n < \mathtt{z}$ and $p_i \leq q$ for all $i$. Hence, the $\longrightarrow_{\mathcal{R}}^{AC}$-normal form of $[|t|]$ is $t' = \mathtt{S}\,p_1\,x_1 \sqcup \ldots \sqcup \mathtt{S}\,p_m\,x_m \sqcup \mathtt{S}\,q\,\mathtt{z}$ with $x_1 < \ldots < x_m < \mathtt{z}$ and $p_i \leq q$, and the $\longrightarrow_{\mathcal{R}}^{AC}$-normal form of $[|u|]$ is $u' = \mathtt{S}\,p'_1\,x'_1 \sqcup \ldots \sqcup \mathtt{S}\,p'_n\,x'_n \sqcup \mathtt{S}\,q'\,\mathtt{z}$ with $x'_1 < \ldots < x'_n < \mathtt{z}$ and $p'_i \leq q'$.

Note also that $t \simeq |t| \simeq [|t|] \simeq t'$, and similarly for $u$ and $u'$.

Hence, if $t' = u'$ then $t \simeq u$.

Conversely, assume that $t \simeq u$. Then, $t' \simeq u'$, and $t'$ and $u'$ have the same canonical form. But a $\longrightarrow_{\mathcal{R}}^{AC}$-normal form $\mathtt{S}\,p_1\,x_1 \sqcup \ldots \sqcup \mathtt{S}\,p_n\,x_n \sqcup \mathtt{S}\,q\,\mathtt{z}$ with $x_1 < \ldots < x_n < \mathtt{z}$ and $p_i \leq q$ is almost a canonical form: it is a canonical form iff $n = 0$ or $p_n < q$. Moreover, if it is not canonical, then $n > 0$ and $p_n = q$, and its canonical form is $\mathtt{S}\,p_1\,x_1 \sqcup \ldots \sqcup \mathtt{S}\,p_n\,x_n$. So, $m = n$ and, for all $i$, $p_i = p'_i$ and $x_i = x'_i$. Moreover, since $t' \simeq u'$, we have $p_n < q$ iff $p'_n < q'$. Therefore, $q = q'$ and $t' = u'$. ◀

▶ Remark. The function mapping every $\mathcal{L}$-term $t$ to the unique $\longrightarrow_{\mathcal{R}}^{AC}$ normal form of $[|t|]$ is not a canonizer in the sense of Shostak as it is not an endofunction. On the other hand, the function mapping every term of $\mathcal{T}$ (guarded terms containing $\mathtt{z}$ with no variable of sort $\mathtt{N}$) to its $\longrightarrow_{\mathcal{R}}^{AC}$ normal form is a canonizer in the sense of Shostak as it satisfies the following properties [26]: (CAN-1) it is idempotent; (CAN-2) it decides $\simeq$ on $\mathcal{T}$; (CAN-3) it preserves variables; (CAN-4) every subterm of a canonical term is canonical; and even (CAN-5) canonization commutes with order-preserving variable renamings.

## 5 Implementation of AC-canonization

To implement AC-canonization in Lambdapi [23], we use an approach introduced in [11]. AC-canonization is done at term construction time. More precisely, we use the mechanism of private data type of OCaml. A private data type is a semi-abstract data type: it is defined as an inductive data type so that users can pattern-match on values of this type but, to build values of this type, one needs to use construction functions. With this mechanism, one can easily enforce some invariant like, here, to have only terms in AC-canonical form. To do so, we only have to replace constructors by construction functions, which is easy and does not require big changes in the code, and implement those construction functions[10]. Moreover, to implement them, we can take advantage of the fact that their arguments are themselves already in AC-canonical form. Finally, note that, by doing so, we get AC-equivalence in the type conversion of Lambdapi for free. On the other hand, we had to slightly adapt the normalization algorithm of Lambdapi [23] to take into account the fact that terms are now put in AC-canonical form after each rewriting step, which may generate new redexes.

[10] See `https://github.com/Deducteam/lambdapi/pull/639`.

## References


1. B. Accattoli and B. Barras. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Conference on Principles and Practice of Declarative Programming*, 2017. doi:10.1145/3131851.3131855.
2. Agda sort system. https://agda.readthedocs.io/en/latest/language/sort-system.html.
3. http://aprove.informatik.rwth-aachen.de/.
4. A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8:1–49, 2012. doi:10.2168/LMCS-8(1:18)2012.
5. A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École Polytechnique, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01235303/.
6. A. Assaf, G. Dowek, J.-P. Jouannaud, and J. Liu. Encoding proofs in Dedukti: the case of Coq proofs, 2016. Presented at the First International Workshop on Hammers for Type Theories (HaTT). URL: https://hal.inria.fr/hal-01330980.
7. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science. Volume 2. Background: computational structures*, pages 117–309. Oxford University Press, 1992.
8. B. Barras, J.-P. Jouannaud, P.-Y. Strub, and Q. Wang. CoqMTU: a higher-order type theory with a predicative hierarchy of universes parameterized by a decidable first-order theory. In *Proceedings of the 26th IEEE Symposium on Logic in Computer Science*, 2011. doi:10.1109/LICS.2011.37.
9. 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.
10. F. Blanqui, G. Dowek, E. Grienenberger, G. Hondet, and F. Thiré. Some axioms for mathematics. In *Proceedings of the 6th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 195, 2021. doi:10.4230/LIPIcs.FSCD.2021.20.
11. F. Blanqui, T. Hardin, and P. Weis. On the implementation of construction functions for non-free concrete data types. In *Proceedings of the 16th European Symposium on Programming*, Lecture Notes in Computer Science 4421, 2007. 15 pages. doi:10.1007/978-3-540-71316-6_8.
12. M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. In *Proceedings of the 1st International Conference on Certified Programs and Proofs*, Lecture Notes in Computer Science 7086, 2011. doi:10.1007/978-3-642-25379-9_26.
13. J. Chrząszcz. Modules in Coq are and will be correct. In *Proceedings of the International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science 3085, 2003. doi:10.1007/978-3-540-24849-1_9.
14. 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.
15. S. Eker. Fast matching in combinations of regular equational theories. In *Proceedings of the 1st International Workshop on Rewriting Logic and Applications*, Electronic Notes in Theoretical Computer Science 4, 1996. doi:10.1016/S1571-0661(04)00035-0.
16. S. Eker. Associative-commutative rewriting on large terms. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2706, 2003. doi:10.1007/3-540-44881-0_3.
17. G. Férey. *Higher-Order Confluence andUniverse Embedding in theLogical Framework*. PhD thesis, Université Paris-Saclay, France, 2021.
18. M. Fernández and J.-P. Jouannaud. Modular termination of term rewriting systems revisited. In *Proceedings of the 10th International Workshop on Specification of Abstract Data Types*, Lecture Notes in Computer Science 906, 1994. doi:10.1007/BFb0014432.

19 G. Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, Université Paris-Saclay, 2020. URL: https://hal.inria.fr/tel-03167579.

20 G. Genestier. Encoding agda programs using rewriting. 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.2020.31.

21 B. Gramlich. Modularity in term rewriting revisited. *Theoretical Computer Science*, 464:3–19, 2012. doi:10.1016/j.tcs.2012.09.008.

22 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.

23 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.

24 D. Kapur and P. Narendran. NP-completeness of the associative-commutative unification and related problems. Unpublished Manuscript. Computer Science Branch, General Electric Corporate Research and Development, Schenectady, NY. See [25], 1986.

25 D. Kapur and P. Narendran. Matching, unification and complexity. *SIGSAM Bull.*, 21(4):6–9, 1987. doi:10.1145/36330.36332.

26 S. Krstić and S. Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2):87–106, 2005. doi:10.1016/j.ic.2004.11.001.

27 C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996. doi:10.1006/jsco.1996.0011.

28 P. Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings of the 1973 Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1975. URL: http://archive-pml.github.io/martin-lof/pdfs/An-Intuitionistic-Theory-of-Types-Predicative-Part-1975.pdf.

29 M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu Matematykow Krajow Slowcanskich, Warszawa, Poland*, 1929.

30 R. Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015. URL: https://pastel.archives-ouvertes.fr/tel-01299180.

31 R. Shostak. Deciding combination of theories. *Journal of the ACM*, 31(1):1–12, 1984.

32 M. Sozeau. Polymorphic universes. https://coq.inria.fr/refman/addendum/universe-polymorphism.html.

33 M. Sozeau and N. Tabareau. Universe polymorphism in Coq. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 8558, 2014. doi:10.1007/978-3-319-08970-6_32.

34 TeReSe. *Term rewriting systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

35 B. Ziliani and M. Sozeau. A comprehensible guide to a new unifier for CIC including universe polymorphism and overloading. *Journal of Functional Programming*, 27(E10), 2017. doi:10.1017/S0956796817000028.

# Adequate and Computational Encodings in the Logical Framework Dedukti

**Thiago Felicissimo** ✉
Université Paris-Saclay, INRIA project Deducteam, Laboratoire de Méthodes Formelles, ENS Paris-Saclay, 91190, France

**Abstract**

DEDUKTI is a very expressive logical framework which unlike most frameworks, such as the Edinburgh Logical Framework (LF), allows for the representation of computation alongside deduction. However, unlike LF encodings, DEDUKTI encodings proposed until now do not feature an adequacy theorem – i.e., a bijection between terms in the encoded system and in its encoding. Moreover, many of them also do not have a conservativity result, which compromises the ability of DEDUKTI to check proofs written in such encodings. We propose a different approach for DEDUKTI encodings which do not only allow for simpler conservativity proofs, but which also restore the adequacy of encodings. More precisely, we propose in this work adequate (and thus conservative) encodings for Functional Pure Type Systems. However, in contrast with LF encodings, ours is computational – that is, represents computation directly as computation. Therefore, our work is the first to present and prove correct an approach allowing for encodings that are both adequate and computational in DEDUKTI.

**2012 ACM Subject Classification** Theory of computation → Type theory; Theory of computation → Equational logic and rewriting

**Keywords and phrases** Type Theory, Logical Frameworks, Rewriting, Dedukti, Pure Type Systems

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.25

**Related Version** *Full Version*: https://arxiv.org/abs/2205.02883

**Acknowledgements** I would like to thank my PhD advisors Frédéric Blanqui and Gilles Dowek for the helpful discussions and comments on this paper. I would also like to thank the anonymous reviewers for their very helpful comments and suggestions.

## 1 Introduction

The research on proof checking naturally leads to the proposal of many logical systems and theories. *Logical frameworks* are a way of addressing this heterogeneity by proposing a common foundation in which systems and theories can be defined. The *Edinburgh Logical Framework* (LF)[17] is one of the milestones in the history of logical frameworks, and proposes the use of a dependently-typed lambda-calculus to express deduction. However, as modern proof assistants move from traditional logics to type theories, where computation plays an important role alongside deduction, it becomes essential for such frameworks to also be able to express computation, something that the LF does not achieve.

The logical framework DEDUKTI[3] addresses this point by extending the LF with rewriting rules, thus allowing for the representation of both deduction and computation. This framework was already proven to be as a very expressive system, and has been used to encode the logics of many proof assistants, such as COQ[14], AGDA[15], PVS[18] and others.

However, an unsatisfying aspect persists as, unlike LF encodings, the DEDUKTI encodings proposed until now are not *adequate*, in the sense that they do not feature a syntactical bijection between the terms of the encoded system and those of the encoding. Such property is key to ensure that the framework faithfully represents the syntax on the encoded system. Moreover, proving that DEDUKTI encodings are *conservative* (i.e., that if the translation of

© Thiago Felicissimo;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 25; pp. 25:1–25:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a type is inhabited, then this type is inhabited) is still a challenge, in particular for recent works such as [14, 23, 15, 18]. This is a problem if one intends to use Dedukti to check the correctness of proofs coming from proof assistants: if conservativity does not hold then the fact that the translation of a proof is checked correct in Dedukti does not imply that this proof is correct.

In the specific case of Pure Type Systems (PTS), a class of type systems which generalizes many others, [9] was the first to propose an encoding of functional PTSs into Dedukti. One of their main contributions is that, contrarily to LF encodings, theirs is *computational* – that is, represents computation in the encoded system directly as computation. The authors then show that the encoding is conservative under the hypothesis of normalization of their rewrite rules.

To address the issue of this unproven assumption, [11] proposed a notion of model of Dedukti and showed using the technique of *reducibility candidates* that the existence of such a model entails the normalization of the encoding. Using this result, the author then showed the conservativity of the encoding of Simple Type Theory and of the Calculus of Constructions. This technique however is not very satisfying as the construction of such models is a very technical task, and needs to be done case by case. One can also wonder why conservativity should rely on normalization.

The cause of this difficulty in [9] and in all other traditional Dedukti encodings comes from a choice made to represent the abstraction and application of the encoded system directly by the abstraction and application of the framework. This causes a confusion as redexes of the encoded system, that represent real computations, get confused with the $\beta$ redexes of the framework, which in other frameworks such as the LF are used exclusively to represent binder substitution. As a non-normal term can contain both types of redexes, it is impossible to inverse translate it as some of these redexes are ill-typed in the original system, and the only way of eliminating these ill-typed redexes is by reducing all of them. One then needs this process to be terminating, which is non-trivial to show as it involves proving that the reduction of the redexes of the encoded system terminates.

The work of [2] first noted this problem and proposed a different approach to show the conservativity of the encoding of PTSs. Instead of relying on the normalization of the encoding, they proposed to directly inverse translate terms without normalizing them. As this creates ill-typed terms, they then used reducibility candidates to show that these ill-typed terms reduce to well-typed ones, thus proving conservativity for the encoding in [9].

Even though this technique is a big improvement over [11], it is still unsatisfying that both of them rely on involved arguments using reducibility, whereas the proofs of LF encodings were very natural. They also both rely on intricate properties of the encoded systems, which is unnatural given that logical frameworks should ideally only require the encoded systems to satisfy some basic properties, and be agnostic with respect to more deep ones – for instance, one should not be obliged to show that a given system is consistent in order to encode it in a logical framework. This reason, coupled with the technicality of these proofs, may explain why recent works such as [23], [18] and [15] have left conservativity as conjecture. Moreover, none of these works have addressed the lack of an adequacy theorem, which until now has remained an overlooked problem in the Dedukti community.

## Our contribution

We propose to depart from the approach of traditional Dedukti encodings by restoring the separation that existed in LF encodings. Our paradigm represents the abstractions and applications of the encoded system not by those of the framework, but by dedicated

constructions. Using this approach, we propose an encoding of functional PTSs that is not only sound and conservative but also adequate. However, in contrast with LF encodings, ours is computational like other Dedukti encodings.

To show conservativity, we leverage the fact that the computational rules of the encoded system are not represented by $\beta$ reduction anymore, but by dedicated rewrite rules. This allows us to normalize only the framework's $\beta$ redexes without touching those associated with the encoded system, and thus performing no computation from its point of view.

To be able to $\beta$ normalize terms, we generalize the proof in [17] to give a general criterion for the normalization of $\beta$ reduction in Dedukti. This criterion imposes rewriting rules to be *arity preserving* (a definition we introduce). This is not satisfied by traditional Dedukti encodings, but poses no problem to ours. The proof uses the simple technique of defining an erasure map into the simply-typed lambda calculus, which is known to be normalizing.

## Outline

We start in Section 2 by recalling the preliminaries about Dedukti. We proceed in Section 3 by proposing a criterion for the normalization of $\beta$ in Dedukti, which is used in our proofs of conservativity and adequacy. In Section 4 we introduce an explicitly-typed version of Pure Type Systems, which is used for the encoding. We then present the encoding in Section 5, and proceed by showing it is sound in Section 6 and that it is conservative and adequate in Section 7. In Section 8 we discuss how our approach can be used together with already known techniques to represent systems with infinitely many sorts. Finally, in Section 9 we discuss more practical aspects by showing how the encoding can be instantiated and used in practice.

# 2 Dedukti

$$\frac{}{\Sigma; - \ \texttt{well-formed}}\ \texttt{Empty} \qquad x \notin \Gamma\ \frac{\Sigma;\Gamma \vdash A : \texttt{TYPE}}{\Sigma;\Gamma, x : A\ \texttt{well-formed}}\ \texttt{Decl}$$

$$c[\Delta] : A \in \Sigma\ \frac{\Sigma;\Delta \vdash A : s \qquad \Sigma;\Gamma \vdash \vec{M} : \Delta}{\Sigma;\Gamma \vdash c[\vec{M}] : A\{\vec{M}\}}\ \texttt{Cons} \qquad \frac{\Sigma;\Gamma\ \texttt{well-formed}}{\Sigma;\Gamma \vdash \texttt{TYPE} : \texttt{KIND}}\ \texttt{Sort}$$

$$x : A \in \Gamma\ \frac{\Sigma;\Gamma\ \texttt{well-formed}}{\Sigma;\Gamma \vdash x : A}\ \texttt{Var} \qquad A \equiv_{\beta\mathcal{R}} B\ \frac{\Sigma;\Gamma \vdash M : A \qquad \Sigma;\Gamma \vdash B : s}{\Sigma;\Gamma \vdash M : B}\ \texttt{Conv}$$

$$\frac{\Sigma;\Gamma \vdash A : \texttt{TYPE} \qquad \Sigma;\Gamma, x : A \vdash B : s}{\Sigma;\Gamma \vdash \Pi x : A.B : s}\ \texttt{Prod} \qquad \frac{\Sigma;\Gamma \vdash M : \Pi x : A.B \qquad \Sigma;\Gamma \vdash N : A}{\Sigma;\Gamma \vdash MN : B\{N/x\}}\ \texttt{App}$$

$$\frac{\Sigma;\Gamma \vdash A : \texttt{TYPE} \qquad \Sigma;\Gamma, x : A \vdash B : s \qquad \Sigma;\Gamma, x : A \vdash M : B}{\Sigma;\Gamma \vdash \lambda x : A.M : \Pi x : A.B}\ \texttt{Abs}$$

**Figure 1** Typing rules for Dedukti.

The logical framework Dedukti [3] has the syntax of the $\lambda$-calculus with dependent types [17] ($\lambda\Pi$-calculus). Like works such as [18], we consider here a version with arities, with the following syntax.

$$A, B, M, N ::= x \mid c[\vec{M}] \mid \texttt{TYPE} \mid \texttt{KIND} \mid MN \mid \lambda x : A.M \mid \Pi x : A.B$$

Here, $c$ ranges in an infinite set of constants $\mathcal{C}$, and $x$ ranges in an infinite set of variables $\mathcal{V}$. Each constant $c$ is assumed to have a fixed arity $n_c$ and for each occurence of $c[\vec{M}]$ we should have $length(\vec{M}) = n_c$. We denote $\Lambda_{\mathtt{DK}}$ the set of terms generated by this grammar. We call a term of the form $\Pi x : A.B$ a *dependent product*, and we write $A \to B$ when $x$ does not appear free in $B$. We allow ourselves sometimes to write $c\ \vec{M}$ instead of $c[\vec{M}]$ to ease the notation.

A *context* $\Gamma$ is a finite sequence of pairs $x : A$ with $A \in \Lambda_{\mathtt{DK}}$. A *signature* $\Sigma$ is a finite set of triples $c[\Delta] : A$ where $A \in \Lambda_{\mathtt{DK}}$ and $\Delta$ is a context containing at least all free variables of $A$. The main difference between Dedukti and the $\lambda\Pi$-calculus is that we also consider a set $\mathcal{R}$ of *rewrite rules*, that is, of pairs of the form $c[\vec{l}] \longrightarrow r$ with $l_1, ..., l_k, r \in \Lambda_{\mathtt{DK}}$. A *theory* is a pair $(\Sigma, \mathcal{R})$ such that all constants appearing in $\mathcal{R}$ are declared in $\Sigma$.

We write $\hookrightarrow_{\mathcal{R}}$ for the context and substitution closure of the rules in $\mathcal{R}$ and $\hookrightarrow_{\beta\mathcal{R}}$ for $\hookrightarrow_{\beta} \cup \hookrightarrow_{\mathcal{R}}$. We also consider the equivalence relation $\equiv_{\beta\mathcal{R}}$ generated by $\hookrightarrow_{\beta\mathcal{R}}$. Finally, we may refer to $\hookrightarrow_{\beta\mathcal{R}}$ and $\equiv_{\beta\mathcal{R}}$ by just $\hookrightarrow$ and $\equiv$.

Typing in Dedukti is given by the rules in Figure 1. In rule `Cons` we use the usual notation $\Sigma; \Gamma \vdash \vec{M} : \Delta$ meaning that $\Delta = x_1 : A_1, ..., x_n : A_n$ and $\Sigma; \Gamma \vdash M_i : A_i\{M_1/x_1\}...\{M_{i-1}/x_{i-1}\}$ is derivable for $i = 1, ..., n$. We then also allow ourselves to write $A\{\vec{M}\}$ instead of $A\{M_1/x_1\}...\{M_n/x_n\}$. We refer to Appendix A for a review of some basic metatheorems.

## 3 Strong Normalization of $\beta$ in Dedukti

In order to show conservativity of encodings, one often needs to be able to $\beta$ normalize terms, thus requiring $\beta$ to be normalizing for well-typed terms. In this section we present a criterion for the normalization of $\beta$ in Dedukti. More precisely, we will see that if $\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is *arity preserving* (a definition we will introduce in this section), then $\beta$ is SN (strongly normalizing) in Dedukti for well-typed terms. The proof generalizes the one given in [17]. However, because of space constraints, we only give the intuition of the proof and refer to the long version in [12] for all the details.

Note that, unlike works such as [8], which provide syntactic criteria on the normalization of $\beta\mathcal{R}$ in Dedukti, we only aim at showing the normalization of $\beta$. In particular $\beta\mathcal{R}$ may not be SN in our setting. Our work has more similar goals to [4], which provides criteria for the SN of $\beta$ in the Calculus of Constructions when adding object-level rewrite rules. However, our work also allows for type-level rewrite rules, which will be needed in our encoding.

Our proof works by defining an erasure map into the simply-typed $\lambda$-calculus, which is known to be SN, and then showing that this map preserves typing and non-termination of $\beta$, thus implying that $\beta$ is SN in Dedukti. Before proceeding, we introduce the following basic definitions.

▶ **Definition 1.**

1. *Given a signature* $\Sigma$*, a constant* $c$ *is type-level (and referred by* $\alpha, \gamma$*) if* $c[\Delta] : A \in \Sigma$ *with* $A$ *of the form* $\Pi\vec{x} : \vec{B}.\mathtt{TYPE}$*, otherwise it is object-level (and referred by* $a, b$*).*
2. *A rewrite rule* $c[\vec{l}] \longrightarrow r$ *is type-level if its head symbol* $c$ *is a type-level constant.*

We can now define the erasure map.

▶ **Definition 2** (Erasure map). *Consider the simple types generated by the grammar*

$$\sigma ::= * \mid \sigma \to \sigma\,.$$

*Moreover, let* $\Gamma_\pi$ *be the context containing for each* $\sigma$ *the declaration* $\pi_\sigma : * \to (\sigma \to *) \to *$*. We define the partial functions* $\|-\|, |-|$ *by the following equations.*

$$\begin{aligned} \|\texttt{TYPE}\| &= * \\ \|\alpha[\vec{M}]\| &= * \\ \|\Pi x : A.B\| &= \|A\| \to \|B\| \\ \|AN\| &= \|A\| \\ \|\lambda x : A.B\| &= \|B\| \end{aligned}$$

$$\begin{aligned} |x| &= x \\ |a[\vec{M}]| &= a\ |\vec{M}| \\ |\alpha[\vec{M}]| &= \alpha\ |\vec{M}| \\ |MN| &= |M||N| \\ |\lambda x : A.M| &= (\lambda z.\lambda x.|M|)|A| \ \textit{where}\ z \notin FV(M) \\ |\Pi x : A.B| &= \pi_{\|A\|}\ |A|\ (\lambda x.|B|) \end{aligned}$$

We also extend the definition of $\|-\|$ (partially) on contexts and signatures by the following equations.

$$\begin{aligned} \|-\| &= - \\ \|x : A, \Gamma\| &= x : \|A\|, \|\Gamma\| \\ \|c[x_1 : A_1, ..., x_n : A_n] : A; \Sigma\| &= (c : \|A_1\| \to ... \to \|A_n\| \to \|A\|), \|\Sigma\| \end{aligned}$$

In order to show the normalization of $\beta$, we need the erasure to preserve typing. The main obstacle when showing this is dealing with the `Conv` rule. To make the proof go through, we would need to show that if $A \equiv B$ then $\|A\| = \|B\|$. In the $\lambda\Pi$-calculus this can be easily shown, however because in DEDUKTI the relation $\equiv$ also takes into account the rewrite rules in $\mathcal{R}$, we can easily build counterexamples in which this does not hold.

▶ **Example 3.** Let $El$ be a type-level constant, and consider the rule

$$El\ (Prod\ A\ B) \longrightarrow \Pi x : El\ A.El\ (B\ x)$$

traditionally used to build DEDUKTI encodings (as in [9]). Note that here we write $\alpha\ \vec{l}$ for $\alpha[\vec{l}]$, to ease the notation. We then have

$$El\ (Prod\ Nat\ (\lambda x.Nat)) \equiv \Pi x : El\ Nat.El\ ((\lambda x.Nat)\ x) \equiv El\ Nat \to El\ Nat$$

but $\|El\ (Prod\ Nat\ (\lambda x.Nat))\| = *$ and $\|El\ Nat \to El\ Nat\| = * \to *$.

If we were to define the arity of a type[1] as the number of consecutive arrows (that is, of $\Pi$s), then we realize that the problem here is that rules such as $El\ (Prod\ A\ B) \longrightarrow \Pi x : El\ A.El\ (B\ x)$ do not preserve the arity. Indeed, $El\ (Prod\ A\ B)$ has arity $0$ because it has no arrows, whereas $\Pi x : El\ A.El\ (B\ x)$ has arity $1$ as it has one arrow[2]. As the left-hand side of a type-level rule always has arity $0$ (because it is of the form $\alpha[\vec{l}]$), to remove these unwanted cases we need for their right-hand sides to also have arity $0$. This motivates the following definition.

▶ **Definition 4** (Arity preserving). *$\mathcal{R}$ is said to be arity-preserving[3] if, for every type-level rewrite rule in $\mathcal{R}$, the right-hand side is in the following grammar, where $\vec{M}, N, A$ are arbitrary.*

$$R ::= \alpha[\vec{M}]\ |\ R\ N\ |\ \lambda x : A.R$$

[1] Note that this concept is different from the notion of arity of constants, as defined in Section 2.
[2] Using a different notation for the dependent product, we can write this type as $(x : El\ A) \to El\ (B\ x)$, which may help to clarify this assertion.
[3] More precisely, this definition also depends on the signature $\Sigma$, as this is used to define which constants are type-level.

It turns out that this condition, together with confluence of $\beta\mathcal{R}$, is enough to show that the translation preserves typing and non-termination. Using the fact that well-typed terms are strongly normalizing in the simply-typed $\lambda$-calculus, we get the following theorem. We refer to the long version in [12] for the proofs.

▶ **Theorem 5** ($\beta$ is SN in Dedukti). *If $\beta\mathcal{R}$ is confluent and $\mathcal{R}$ is arity-preserving, then $\beta$ is strongly normalizing for well-typed terms in Dedukti.*

## 4 Pure Type Systems

Pure type systems (or PTSs) is a class of type systems that generalizes many other systems, such as the Calculus of Constructions and System F. They are parameterized by a set of *sorts* $\mathcal{S}$ (refereed to by the letter $s$) and two relations $\mathcal{A} \subseteq \mathcal{S}^2, \mathcal{R} \subseteq \mathcal{S}^3$. In this work we restrict ourselves to functional PTSs, for which $\mathcal{A}$ and $\mathcal{R}$ are functional relations. This restriction covers almost all of PTSs used in practice, and gives a much more well behaved metatheory.

In this paper we consider a variant of PTSs with explicit parameters. That is, just like when taking the projection of a pair $\pi^1(p)$ we can make explicit all parameters and write $\pi^1(A, B, p)$ where $p : A \times B$, we can also write $\lambda(A, [x]B, [x]M)$ instead of $\lambda x : A.M$ and $@(A, [x]B, M, N)$ instead of $MN$. Moreover, if $(-) \times (-)$ is a universe-polymorphic definition, we should also write $\pi^1_{s_A,s_B}(A, B, p)$ to make explicit the sort parameters. As in PTSs the dependent product is used across multiple sorts, we then should also write $\lambda_{s_A,s_B}(A, [x]B, [x]M)$, $@_{s_A,s_B}(A, [x]B, M, N)$ and $\Pi_{s_A,s_B}(A, [x]B)$. To be more direct, we render explicit the parameters on the dependent product type and on its constructor (abstraction) and eliminator (application). Because of this interpretation in which we are rendering the parameters of $\lambda$ and $@$ explicit, we name this version of PTSs as Explicitly-typed Pure Type Systems (EPTSs).

Reduction is then defined by the context closure of the $\beta$ rules[4]

$$@_{s_1,s_2}(A, [x]B, \lambda_{s_1,s_2}(A', [x]B', [x]M), N) \longrightarrow M\{N/x\}$$

given for each $(s_1, s_2, s_3) \in \mathcal{R}$. Typing is given by the rules in Figure 2.

$$\frac{}{-\text{ well-formed}}\ \textsc{Empty} \qquad x \notin \Gamma\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \text{ well-formed}}\ \textsc{Decl}$$

$$A \equiv B\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \textsc{Conv} \qquad (s_1, s_2) \in \mathcal{A}\ \frac{\Gamma \text{ well-formed}}{\Gamma \vdash s_1 : s_2}\ \textsc{Sort}$$

$$x : A \in \Gamma\ \frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A}\ \textsc{Var} \qquad (s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{s_1,s_2}(A, [x]B) : s_3}\ \textsc{Prod}$$

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda_{s_1,s_2}(A, [x]B, [x]M) : \Pi_{s_1,s_2}(A, [x]B)}\ \textsc{Abs}$$

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma \vdash N : A \qquad \Gamma \vdash M : \Pi_{s_1,s_2}(A, [x]B)}{\Gamma \vdash @_{s_1,s_2}(A, [x]B, M, N) : B\{N/x\}}\ \textsc{App}$$

**Figure 2** Typing rules for Explicitly-typed Pure Type Systems.

[4] We consider a linearized variant of the expected non-left linear rule $@_{s_1,s_2}(A, [x]B, \lambda_{s_1,s_2}(A, [x]B, [x]M), N) \longrightarrow M\{N/x\}$, which is non-confluent in untyped terms. By linearizing it, we get a much more well-behaved rewriting system, where confluence holds for all terms. Moreover, whenever the left hand side is well-typed, the typing constraints impose $A \equiv A'$ and $B \equiv B'$.

This modification is just a technical change that will help us during the translation, as our encoding needs the data of such parameters often left implicit. Other works such as [20] and [21] also consider similar variants, though none of them corresponds exactly to ours. Therefore, we had to develop the basic metatheory of our version in [13], and we have found that the usual meta-theoretical properties of functional PTSs are preserved when moving to the explicitly-typed version (see Appendix B). More importantly, by a proof that uses ideas present in [21], we have shown the following equivalence.

Let $|-|$ be the erasure map defined in the most natural way from an EPTS to its corresponding PTS. Moreover, for a system $X$ let $\Lambda(\Gamma \vdash_X _ : A)$ be the set of $M \in \Lambda_X$ with $\Gamma \vdash_X M : A$. Finally, let $\equiv_I$ be defined by $M \equiv_I N$ iff $|M| = |N|$ and $M \equiv N$.

▶ **Theorem 6** (Equivalence between PTSs and EPTSs[13]). *Consider a functional PTS. If* $\Gamma \vdash_{PTS} A$ *type, then there are* $\Gamma', A'$ *with* $|\Gamma'| = \Gamma, |A'| = A$ *such that we have a bijection*

$$\Lambda(\Gamma \vdash_{PTS} _ : A) \simeq \Lambda(\Gamma' \vdash_{EPTS} _ : A')/\equiv_I$$

## 5 Encoding EPTSs in Dedukti

This section presents our encoding of functional EPTSs in Dedukti. In order to ease the notation, from now one we write $c\ \vec{M}$ for $c[\vec{M}]$. The basis for the encoding is given by a theory $(\Sigma_{\mathbf{EPTS}}, \mathcal{R}_{\mathbf{EPTS}})$ which we will construct step by step here.

Pure Type Systems (explicitly-typed or not) feature two kinds of types: dependent products and universes. We start by building the representation of the latter. For each $s \in \mathcal{S}$ we declare a type $U_s$ to represent the type of elements of $s$. However, as the terms $A$ with $\Gamma \vdash_{EPTS} A : s$ are themselves types, we also need to declare a function $El_s$ which maps each such $A$ to its corresponding type. As for each $(s_1, s_2) \in \mathcal{A}$ we have $\vdash_{EPTS} s_1 : s_2$, we also declare a constant $u_{s_1}$ in $U_{s_2}$ to represent this. Finally, as the sorts $s_1$ with $(s_1, s_2) \in \mathcal{A}$ now can be represented by both $U_{s_1}$ and $El_{s_2}\ u_{s_1}$, we add a rewrite rule to identify these representations. This encoding resembles the definition of universes in type theories *à la* Tarski, and also follows traditional representations of universes in Dedukti as in [9].

$$U_s : \mathtt{TYPE} \qquad El_s[A : U_s] : \mathtt{TYPE} \qquad \text{for } s \in \mathcal{S}$$

$$u_{s_1} : U_{s_2} \qquad El_{s_2}\ u_{s_1} \hookrightarrow_{u_{s_1}\text{-red}} U_{s_1} \qquad \text{for } (s_1, s_2) \in \mathcal{A}$$

We now move to the representation of the dependent product type. We first declare a constant to represent the type formation rule for the dependent product.

$$Prod_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}] : U_{s_3} \qquad \text{for } (s_1, s_2, s_3) \in \mathcal{R}$$

Traditional Dedukti encodings would normally continue here by introducing the rule $El_{s_3}\ (Prod_{s_1,s_2}\ A\ B) \longrightarrow \Pi x : El_{s_1}\ A. El_{s_2}\ (B\ x)$, identifying the dependent product of the encoded theory with the one of Dedukti, thus allowing for the use of the framework's abstraction, application and $\beta$ to represent the ones of the encoded system. We instead keep them separate and declare constants representing the introduction and elimination rules for the dependent product being encoded, that is, representing abstraction and application.

$$abs_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}; M : \Pi x : El_{s_1}\ A. El_{s_2}\ (B\ x)] : El_{s_3}(Prod_{s_1,s_2}\ A\ B)$$
$$app_{s_1,s_2}[A : U_{s_1}; B : El_{s_1}\ A \to U_{s_2}; M : El_{s_3}(Prod_{s_1,s_2}\ A\ B); N : El_{s_1}\ A] : El_{s_2}(B\ N)$$
$$app_{s_1,s_2}\ A\ B\ (abs_{s_1,s_2}\ A'\ B'\ M)\ N \hookrightarrow_{beta_{s_1,s_2}} M\ N \qquad \text{for } (s_1, s_2, s_3) \in \mathcal{R}$$

We note that this idea is also hinted in [1], though they did not pursue it further. This approach also resembles the one of the Edinburgh Logical Framework (LF) [17] in which the framework's abstraction is used exclusively for binding. We are however able to encode computation directly as computation with the rule $beta_{s_1,s_2}$, whereas the LF handles computation by encoding it as an equality judgment, thus introducing explicit coercions in the terms. Some other variants such as [16] prevent the introduction of such coercions, but computation is still represented by an equality judgment instead of being represented by computation.

This finishes the definition of the theory $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$. Now we ready to define the translation function $[\![-]\!]$.

$$[\![x]\!] = x$$
$$[\![s]\!] = u_s$$
$$[\![\Pi_{s_1,s_2}(A, [x]B)]\!] = Prod_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])$$
$$[\![\lambda_{s_1,s_2}(A, [x]B, [x]M)]\!] = abs_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ (\lambda x : El_{s_1}\ [\![A]\!].[\![M]\!])$$
$$[\![@_{s_1,s_2}(A, [x]B, M, N)]\!] = app_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!])\ [\![M]\!]\ [\![N]\!]$$

We also extend $[\![-]\!]$ to well-formed contexts by the following definition. Note that because we are dealing with functional EPTSs, the sort of $A$ in $\Gamma$ is unique, hence the following definition makes sense.

$$[\![-]\!] = -$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : El_{s_A}\ [\![A]\!] \quad \text{where } \Gamma \vdash A : s_A$$

▶ Remark 7. Note that in the definition of $[\![-]\!]$ it was essential for $\lambda$ and $@$ to make explicit the types $A$ and $B$, as the constants $abs_{s_1,s_2}$ and $app_{s_1,s_2}$ require their translations. Had we had for instance just $\lambda x : A.M$, we could then make the translation dependent on $\Gamma$ and take a $B$ such that $\Gamma, x : A \vdash M : B$. However, because $[\![-]\!]$ is defined by induction and $B$ is not a subterm of $\lambda x : A.M$, we cannot apply $[\![-]\!]$ to $B$. Therefore, when doing an encoding in Dedukti one should first render explicit the needed data before translating, and then show an equivalence theorem between the explicit and implicit versions (in our case, Theorem 6).

Moreover, note that by also making the sorts explicit in $\lambda_{s_1,s_2}, @_{s_1,s_2}, \Pi_{s_1,s_2}$ our translation can be defined purely syntactically. If this information were not in the syntax, we could still define $[\![-]\!]$ by making it dependent on $\Gamma$, as is usually done with traditional Dedukti encodings[9]. Nevertheless, this complicates many proofs, as each time we apply $[\![-]\!]_\Gamma$ to a term we need to know it is well-typed in $\Gamma$. Moreover, properties which should concern all untyped terms (such as preservation of computation) would then be true only for well-typed ones.

In order to understand more intuitively how the encoding works, let's look at an example.

▶ **Example 8.** Recall that System F can be defined by the sort specification $\mathcal{S} = \{Type, Kind\}, \mathcal{A} = \{(Type, Kind)\}, \mathcal{R} = \{(Type, Type, Type), (Kind, Type, Type)\}$. In this EPTS, we can express the polymorphic identity function, traditionally written as $\lambda A : Type.\lambda x : A.x$, by

$$\lambda_{Kind,Type}(Type, [A]\Pi_{Type,Type}(A, [x]A), [A]\lambda_{Type,Type}(A, [x]A, [x]x))$$

This term is represented in our encoding by

$$abs_{Kind,Type}\ u_{Type}\ (\lambda A.Prod_{Type,Type}\ A\ (\lambda x.A))\ (\lambda A.abs_{Type,Type}\ A\ (\lambda x.A)\ (\lambda x.x))$$

where we omit the type annotations in the abstractions, to improve readability.

## 6 Soundness

An encoding is said to be sound when it preserves the typing relation of the original system. In this section we will see that our encoding has this fundamental property. We start by establishing some conventions in order to ease notations.

▶ **Convention 9.** *We establish the following notations.*

- *We write* $\Sigma; \Gamma \vdash_{\mathsf{DK}} M : A$ *for a* Dedukti *judgment and* $\Gamma \vdash M : A$ *for an EPTS judgment*
- *As the same signature* $\Sigma_{\mathsf{EPTS}}$ *is used everywhere, when referring to* $\Sigma_{\mathsf{EPTS}}; \Gamma \vdash_{\mathsf{DK}} M : A$ *we omit it and write* $\Gamma \vdash_{\mathsf{DK}} M : A$.

Before showing soundness, we start by establishing some basic results.

▶ **Proposition 10** (Basic properties)**.** *We have the following basic properties.*

1. *Confluence: The rewriting rules of the encoding are confluent with* $\beta$.
2. *Well-formedness of the signature: For all* $c[\Delta] : A \in \Sigma_{\mathsf{EPTS}}$*, we have* $\Delta \vdash_{\mathsf{DK}} A : s$.
3. *Subject reduction for* $\beta$*: If* $\Gamma \vdash_{\mathsf{DK}} M : A$ *and* $M \longrightarrow_\beta M'$ *then* $\Gamma \vdash_{\mathsf{DK}} M' : A$.
4. *Strong normalization for* $\beta$*: If* $\Gamma \vdash_{\mathsf{DK}} M : A$*, the* $\beta$ *is strongly normalizing for* $M$.
5. *Compositionality: For all* $M, N \in \Lambda_{EPTS}$ *we have* $[\![M]\!]\{[\![N]\!]/x\} = [\![M\{N/x\}]\!]$.

**Proof.**

1. The considered rewrite rules form an orthogonal combinatory reduction system, and therefore are confluent[19].
2. Can be shown for instance with LambdaPi[10], an implementation of Dedukti.
3. Subject reduction of $\beta$ is implied by confluence of $\beta\mathcal{R}_{\mathsf{EPTS}}$[6].
4. $\mathcal{R}_{\mathsf{EPTS}}$ is arity preserving and $\beta\mathcal{R}_{\mathsf{EPTS}}$ is confluent, thus *$\beta$ is SN in Dedukti* (Theorem 5) applies.
5. By induction on $M$. ◀

▶ Remark 11. We could also show subject reduction of our encoding, either using the method in [7] or LambdaPi[10]. However, we will see that our proof does not actually require subject reduction of $\mathcal{R}_{\mathsf{EPTS}}$. Therefore, we conjecture that our proof method can also be adapted to systems that do not satisfy subject reduction.

▶ **Lemma 12** (Preservation of computation)**.** *Let* $M, N \in \Lambda_{EPTS}$. *We have*

1. $M \longrightarrow N$ *implies* $[\![M]\!] \longrightarrow^* [\![N]\!]$
2. $M \equiv N$ *implies* $[\![M]\!] \equiv [\![N]\!]$

**Proof.** Intuitively, the first part holds because a $\beta$ step in the source system is represented by a *beta* step followed by a $\beta$ step in Dedukti. It is shown by induction on the rewriting context, using compositionality of $[\![-]\!]$ for the base case. The second part follows by induction on $\equiv$ and uses part 1. ◀

Recall that a sort $s \in \mathcal{S}$ is said to be a top-sort if there is no $s'$ with $(s, s') \in \mathcal{A}$. The following auxiliary lemma allows us to switch between sort representations and is heavily used in the proof of soundness.

▶ **Lemma 13** (Equivalence for sort representations)**.** *If* $s$ *is not a top-sort, then*

$$\Gamma \vdash_{\mathsf{DK}} M : U_s \iff \Gamma \vdash_{\mathsf{DK}} M : El_{s'}\ u_s$$

*where* $(s, s') \in \mathcal{A}$.

With all these results in hand, we can now show the soundness of our encoding.

▶ **Theorem 14** (Soundness). *Let $\Gamma$ be a context and $M, A$ terms in an EPTS. We have*

- *If* $\Gamma$ well-formed *then* $[\![\Gamma]\!]$ `well-formed`
- *If* $\Gamma \vdash M : A$ *then*
  - *if $A$ is a top-sort then* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : U_A$
  - *else* $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![M]\!] : El_{s_A}\ [\![A]\!]$*, where* $\Gamma \vdash A : s_A$

**Proof.** By structural induction on the proof of the judgment. We present here only the case Prod to show the idea, the other cases are detailed in the long version in [12].

**Case Prod.** The proof ends with

$$(s_1, s_2, s_3) \in \mathcal{R}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{s_1,s_2}(A, [x]B) : s_3}\ \textsc{Prod}$$

By the IH and Lemma 13, we have $[\![\Gamma]\!] \vdash_{\mathtt{DK}} [\![A]\!] : U_{s_1}$ and $[\![\Gamma]\!], x : El_{s_1}\ [\![A]\!] \vdash_{\mathtt{DK}} [\![B]\!] : U_{s_2}$. By `Abs` we get $[\![\Gamma]\!] \vdash_{\mathtt{DK}} \lambda x : El_{s_1}\ [\![A]\!].[\![B]\!] : El_{s_1}\ [\![A]\!] \to U_{s_2}$, therefore it suffices to apply `Cons` with $Prod_{s_1,s_2}$ to conclude

$$[\![\Gamma]\!] \vdash_{\mathtt{DK}} Prod_{s_1,s_2}\ [\![A]\!]\ (\lambda x : El_{s_1}\ [\![A]\!].[\![B]\!]) : U_{s_3}$$

If $s_3$ is not a top-sort, we then apply Lemma 13. ◀

## 7 Conservativity and Adequacy

Many works proposing Dedukti encodings often stop after showing soundness and leave conservativity as a conjecture. This is because, when mixing the rules $\beta$ with *beta*, as done in traditional Dedukti encodings, one needs to show the termination of both, given that to show conservativity one often considers terms in normal form [9] (with the notable exception of [2]). However this problem is non-trivial, and in particular the normalization of $\beta \cup$ *beta* implies the termination (and thus normally also the consistency) of the encoded system. This is also unnatural, as logical frameworks should be agnostic to the fact that a system is consistent or not, and thus this should not be required to show conservativity.

In this section we will show how conservativity can be proven without difficulties when we distinguish the rules $\beta$ and *beta*. In particular, our proof does not need $\beta \cup$ *beta* to be normalizing, and thus also applies to non-normalizing and inconsistent systems.

We start by defining a notion of invertible forms and an inverse translation which allows to invert them into the original system. After proving some basic properties about them, we then proceed with the proof of conservativity.

### 7.1 The inverse translation

▶ **Definition 15** (Invertible forms). *We call the terms generated by the following grammar the invertible forms. The $s_i$ are arbitrary sorts in $\mathcal{S}$, whereas the $T_1, T_2$ are arbitrary terms.*

$$\begin{aligned} M, N, A, B ::= {} & x \mid u_s \mid abs_{s_1,s_2}\ A\ (\lambda x : T_1.B)\ (\lambda x : T_2.M) \mid (\lambda x : T.M)\ N \\ & \mid Prod_{s_1,s_2}\ A\ (\lambda x : T_1.B) \mid app_{s_1,s_2}\ A\ (\lambda x : T_1.B)\ M\ N \end{aligned}$$

Note that this definition includes some terms which are not in $\beta$ normal form. The next definition justifies the name of invertible forms: we know how to invert them.

▶ **Definition 16.** *We define the inverse translation function* $|-| : \Lambda_{\mathrm{DK}} \to \Lambda_{EPTS}$ *on invertible forms by structural induction.*

$$|x| = x \qquad |Prod_{s_1,s_2}\ A\ (\lambda x : _.B)| = \Pi_{s_1,s_2}(|A|, {}_{[x]}|B|)$$
$$|u_s| = s \qquad |abs_{s_1,s_2}\ A\ (\lambda x : _.B)\ (\lambda x : _.M)| = \lambda_{s_1,s_2}(|A|, {}_{[x]}|B|, {}_{[x]}|M|)$$
$$|(\lambda x : _.M)\ N| = |M|\{|N|/x\} \qquad |app_{s_1,s_2}\ A\ (\lambda x : _.B)\ M\ N| = @_{s_1,s_2}(|A|, {}_{[x]}|B|, |M|, |N|)$$

We can show, as expected, that the terms in the image of the translation $[\![-]\!]$ are invertible forms and that $|-|$ is a left inverse of $[\![-]\!]$. The proof is a simple induction on $M$.

▶ **Proposition 17.** *For all* $M \in \Lambda_{EPTS}$, $[\![M]\!]$ *is an invertible form and* $|[\![M]\!]| = M$.

The following lemma shows that invertible forms are closed under rewriting and that this rewriting can also be inverted into the EPTS.

▶ **Proposition 18.** *Let* $M$ *be an invertible form.*

1. *If* $N$ *is an invertible form, then* $M\{N/x\}$ *is also and* $|M|\{|N|/x\} = |M\{N/x\}|$.
2. *If* $M \hookrightarrow_{beta_{s_1,s_2}} N$ *then* $N$ *is an invertible form and* $|M| \hookrightarrow^*_\beta |N|$
3. *If* $M \hookrightarrow_{\beta,u_{s_1}\text{-}red} N$ *then* $N$ *is an invertible form and* $|M| = |N|$.
4. *If* $M \hookrightarrow^* N$ *then* $N$ *is an invertible form and* $|M| \hookrightarrow^* |N|$.

**Proof.** The property 1 is shown by induction on $M$, whereas 2, 3 follow by induction on the rewrite context and 4 follows directly from 2, 3. ◀

▶ Remark 19. Note that this last proposition explains the difference between the $\beta$ and $beta_{s_1,s_2}$ steps. Whereas $beta_{s_1,s_2}$ steps represent the real computation steps that take place in the encoded system, $\beta$ steps are invisible because they correspond to the framework's substitution, an administrative operation that is implicit in the encoded system. Therefore, it was expected that $beta_{s_1,s_2}$ steps would be reflected into the original system, whereas $\beta$ steps would be silent.

Putting all this together, we deduce that computation and conversion in Dedukti are reflected in the encoded system.

▶ **Corollary 20** (Reflection of computation)**.** *For* $M, N \in \Lambda_{EPTS}$, *we have*

1. *If* $[\![M]\!] \hookrightarrow^* [\![N]\!]$ *then* $M \hookrightarrow^* N$.
2. *If* $[\![M]\!] \equiv [\![N]\!]$ *then* $M \equiv N$.

**Proof.**

1. Immediate consequence of Proposition 18 and Proposition 17.
2. Follows from confluence of $\beta\mathcal{R}_{\mathrm{EPTS}}$ and also Proposition 18 and Proposition 17. ◀

Note that for part 2 we really need $\beta\mathcal{R}_{EPTS}$ to be confluent. Indeed, If $[\![M]\!] \longleftrightarrow N$ then we cannot apply $|-|$ to $N$ because it might not be an invertible form.

## 7.2 Conservativity

Before showing conservativity, we show the following auxiliary result, saying that every $\beta$ normal term $M$ that has type $\Pi x : A.B$ in $[\![\Gamma]\!]$ is an abstraction.

▶ **Lemma 21.** *Let* $M$ *be in* $\beta$*-normal form. If* $[\![\Gamma]\!] \vdash_{\mathrm{DK}} M : \Pi x : A.B$ *then* $M = \lambda x : A'.N$ *with* $A' \equiv A$ *and* $[\![\Gamma]\!], x : A \vdash_{\mathrm{DK}} N : B$.

**Proof.** By induction on $M$. $M$ cannot be a variable or constant, as there is no $x : C \in [\![\Gamma]\!]$ or $c[\Delta] : C \in \Sigma_{\texttt{EPTS}}$ with $C \equiv \Pi x : A.B$. If $M = M_1 M_2$, then $M_1$ has a type of the form $\Pi x' : A'.B'$. By IH we get that $M_1$ is an abstraction, which contradicts the fact that $M$ is in $\beta$ normal form.

Therefore, $M$ is an abstraction, of the form $M = \lambda x : A'.N$. By inversion of typing, we thus have $[\![\Gamma]\!], x : A' \vdash_{\texttt{DK}} N : B'$ with $A' \equiv A$ and $B' \equiv B$. We can then use *Conv in context for DK* (Theorem 30) and `Conv` to derive $[\![\Gamma]\!], x : A \vdash_{\texttt{DK}} N : B$. ◀

We are now ready to show conservativity for $\beta$ normal forms. However, if we also want to show adequacy later, we also need to show that $|-|$ is a kind of right inverse to $[\![-]\!]$. But because the inverse translation does not capture the information in the type annotations of binders, $[\![|M|]\!] = M$ does not hold.

▶ **Example 22.** Take any invertible forms $A, B$ and a term $T$ with $T \neq El_{s_1}\ A$. Then the term $M = Prod_{s_1,s_2}\ A\ (\lambda x : T.B)$ is sent by $|-|$ into $\Pi_{s_1,s_2}(|A|, [x]|B|)$, which is then sent by $[\![-]\!]$ into $Prod_{s_1,s_2}\ [\![|A|]\!]\ (\lambda x : El_{s_1}\ [\![|A|]\!].[\![|B|]\!])$. Therefore, even if have $[\![|B|]\!] = B$ and $[\![|A|]\!] = A$, we still have $T \neq El_{s_1}\ A$, implying $M \neq [\![|M|]\!]$. However, if $M$ is typable, then by typing constraints we should nevertheless have $T \equiv El_{s_1}\ A$.

Therefore, while proving conservativity we will show a weaker property: for the well-typed terms we are interested in, $|-|$ is a right inverse up to the following "hidden" conversion.

▶ **Definition 23** (Hidden step). *We say that a rewriting step $M \longrightarrow N$ is hidden when it happens on the type annotation of a binder. More formally, we should have a rewriting context $C(-)$ and terms $A, A', P$ such that $A \longrightarrow A'$, $M = C(\lambda x : A.P)$ and $N = C(\lambda x : A'.P)$. We denote the conversion generated by such rules by $\equiv_H$.*

We now have all ingredients to show that the encoding is conservative for $\beta$ normal forms.

▶ **Theorem 24** (Conservativity of $\beta$ normal forms). *Suppose $\Gamma \vdash A$ type and let $M \in \Lambda_{\texttt{DK}}$ be a $\beta$ normal form such that $[\![\Gamma]\!] \vdash_{\texttt{DK}} M : T$, with $T = El_{s_A}\ [\![A]\!]$ or $T = U_A$. Then $M$ is an invertible form, $\Gamma \vdash |M| : A$ and $[\![|M|]\!] \equiv_H M$.*

**Proof.** By induction on $M$.

**Case $M = \lambda x : A'.M'$:** By inversion we have $M : \Pi x : A'_1.A'_2$ with $T \equiv \Pi x : A'_1.A'_2$. This then implies that $T$ reduces to a dependent product, but because $T$ is of the form $El_{s_A}\ [\![A]\!]$ or $U_A$ and $\mathcal{R}_{\texttt{EPTS}}$ is arity preserving, this cannot hold. Thus, this case is impossible.

**Case $M = M_1 M_2$:** As $M$ is in beta normal form, its head symbol is a constant or variable. However, there is no $c[\Delta] : C \in \Sigma_{\texttt{EPTS}}$ or $x : C \in \Gamma$ with $C\{\vec{M}\}$ convertible to a dependent product type, for some $\vec{M}$. Hence, this case is impossible.

**Case $M = x$:** If $M = x$, by inversion of typing there is $x : El_{s_B}\ [\![B]\!] \in [\![\Gamma]\!]$ with $T \equiv El_{s_B}\ [\![B]\!]$. Therefore, we deduce $A \equiv B$ and thus we can derive $\Gamma \vdash x : A$ by applying Var with $x : B \in \Gamma$, then Conv with $A \equiv B$ and $\Gamma \vdash A$ *type*.

**Case $M = c[\vec{M}]$:** We proceed by case analysis on $c$. We present only case $c = Prod_{s_1,s_2}$ here and refer to the long version in [12] for all the details.

▶ Note 25. In the following, to improve readability we omit the typing hypothesis when applying `Conv`. However, all such uses can be justified.

**Case $c = Prod_{s_1,s_2}$:** By inversion of typing, we have

1. $\vec{M} = M_1\ M_2$
2. $[\![\Gamma]\!] \vdash_{\texttt{DK}} M_1 : U_{s_1}$
3. $[\![\Gamma]\!] \vdash_{\texttt{DK}} M_2 : El_{s_1}\ M_1 \to U_{s_2}$
4. $T \equiv U_{s_3}$

As $M_1$ is in $\beta$ normal form, by IH $M_1$ is an invertible form, $\Gamma \vdash |M_1| : s_1$ and $[\![|M_1|]\!] \equiv_H M_1$.

By Lemma 21 applied to 3, we get $M_2 = \lambda x : B.N$ and $B \equiv El_{s_1}\ M_2$ with $[\![\Gamma]\!], x : El_{s_1}\ M_1 \vdash N : U_{s_2}$. Because $M_1 \equiv [\![|M_1|]\!]$, we have $[\![\Gamma]\!], x : El_{s_1}\ [\![|M_1|]\!] \vdash N : U_{s_2}$. As $\Gamma \vdash |M_1| : s_1$ we have $\Gamma, x : |M_1|$ well-formed and thus by IH $N$ is an invertible form and we have $[\![|N|]\!] \equiv_H N$ and $\Gamma, x : |M_1| \vdash |N| : s_2$.

Therefore, by PROD we have $\Gamma \vdash \Pi_{s_1,s_2}(|M_1|, [x]|N|) : s_3$, and then by CONV with $A \equiv s_3$ we conclude $\Gamma \vdash \Pi_{s_1,s_2}(|M_1|, [x]|N|) : A$. Finally, as $M_1 \equiv_H [\![|M_1|]\!]$, $N \equiv_H [\![|N|]\!]$ and $B \equiv El_{s_1}\ M_1 \equiv El_{s_1}\ [\![|M_1|]\!]$, we conclude

$$\begin{aligned} M &= Prod_{s_1,s_2}\ M_1\ M_2 = Prod_{s_1,s_2}\ M_1\ (\lambda x : B.N) \\ &\equiv_H Prod_{s_1,s_2} [\![|M_1|]\!]\ (\lambda x : El_{s_1}\ [\![|M_1|]\!].[\![|N|]\!]) = [\![\Pi_{s_1,s_2}(|M_1|, [x]|N|)]\!] = [\![|M|]\!] \end{aligned}$$ ◀

By *Basic properties* (Proposition 10), $\beta$ is strongly normalizing and type preserving. Therefore from the previous result we can immediately get full conservativity.

▶ **Theorem 26** (Conservativity). *Let $\Gamma \vdash A$ type, $M \in \Lambda_{\text{DK}}$ such that $[\![\Gamma]\!] \vdash_{\text{DK}} M : T$, with $T = El_{s_A}\ [\![A]\!]$ or $T = U_A$. We have $\Gamma \vdash |NF_\beta(M)| : A$ and $M \hookrightarrow^*_\beta NF_\beta(M) \equiv_H [\![|NF_\beta(M)|]\!]$.*

Note that this also gives us a straightforward algorithm to invert terms: it suffices to normalize with $\beta$ and then apply $|-|$.

## 7.3 Adequacy

If we write $\Lambda(\Gamma \vdash_{EPTS} _ : A)$ for the set of $M \in \Lambda_{EPTS}$ such that $\Gamma \vdash M : A$ and $\Lambda_{NF}(\Gamma \vdash_{\text{DK}} _ : T)$ for the set of $M \in \Lambda_{\text{DK}}$ in $\beta$ normal form such that $\Gamma \vdash_{\text{DK}} M : T$, we can show our adequacy theorem. This result follows by simply putting together *Basic properties* (Proposition 10), *Preservation of computation* (Lemma 12), *Soundness* (Theorem 14), *Reflection of computation* (Corollary 20) and *Conservativity* (Theorem 26).

▶ **Theorem 27** (Computational adequacy). *For $A, \Gamma$ with $\Gamma \vdash A$ type, let $T = U_A$ if $A$ is a top sort, otherwise $T = El_{s_A}\ [\![A]\!]$. We have a bijection*

$$\Lambda(\Gamma \vdash_{EPTS} _ : A) \simeq \Lambda_{NF}([\![\Gamma]\!] \vdash_{\text{DK}} _ : T)/\equiv_H$$

*given by $[\![-]\!]$ and $|-|$. It is compositional in the sense that $[\![-]\!]$ commutes with substitution. It is computational in the sense that $M \hookrightarrow^* N$ iff $[\![M]\!] \hookrightarrow^* [\![N]\!]$. Moreover, any $M$ satisfying $[\![\Gamma]\!] \vdash_{\text{DK}} M : T$ has such a $\beta$ normal form.*

# 8 Representing systems with infinitely many sorts

We have presented an encoding of EPTSs in DEDUKTI that is sound, conservative and adequate. However when using it in practice with DEDUKTI implementations we run into problems when representing systems with infinitely many sorts, such as in Martin-Löf's Type Theory or the Extended Calculus of Constructions. Indeed, in this case our encoding needs an infinite number of constant and rule declarations, which cannot be made in practice.

One possible solution is to approximate the infinite sort structure by a finite one. Indeed, every proof in an infinite sort systems only uses a finite number of sorts, and thus does not need all of them to be properly represented.

A different approach proposed in [1] is to internalize the indices of $Prod_{s_1,s_2}$, $El_{s_1}$, ... and represent them inside DEDUKTI. In order to apply this method, we chose to stick with systems in which $\mathcal{A}, \mathcal{R}$ are total functions $\mathcal{S} \to \mathcal{S}$ and $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ respectively. Note that this is true for almost all infinite sort systems used in practice, and this will greatly simplify our presentation.

We can now declare a constant $\widehat{\mathcal{S}}$ to represent the type of sorts in $\mathcal{S}$ and two constants $\widehat{\mathcal{A}}, \widehat{\mathcal{R}}$ to represent the functions $\mathcal{A}, \mathcal{R}$. Then, each of our previously declared families of constants now becomes a single one, by taking arguments of type $\widehat{\mathcal{S}}$. The same happens with the rewrite rules. This leads to the theory presented in Figure 3, which we call $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$.

$$\widehat{\mathcal{S}} : \texttt{TYPE}$$
$$\widehat{\mathcal{A}}[s_1 : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$$
$$\widehat{\mathcal{R}}[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$$

$$U[s : \widehat{\mathcal{S}}] : \texttt{TYPE}$$
$$El[s : \widehat{\mathcal{S}}; A : U\ s] : \texttt{TYPE}$$
$$u[s : \widehat{\mathcal{S}}] : U\ (\widehat{\mathcal{A}}\ s)$$
$$El\ s'\ (u\ s) \longrightarrow_{u\text{-red}} U\ s$$

$$Prod[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2] : U\ (\widehat{\mathcal{R}}\ s_1\ s_2)$$
$$abs[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2; N : \Pi x : El\ s_1\ A. El\ s_2\ (B\ x)] : El\ (\widehat{\mathcal{R}}\ s_1\ s_2)\ (Prod\ s_1\ s_2\ A\ B)$$
$$app[s_1 : \widehat{\mathcal{S}}; s_2 : \widehat{\mathcal{S}}; A : U\ s_1; B : El\ s_1\ A \to U\ s_2; M : El\ (\widehat{\mathcal{R}}\ s_1\ s_2)\ (Prod\ s_1\ s_2\ A\ B); N : El\ s_1\ A] : El\ s_2\ (B\ N)$$
$$app\ s_1\ s_2\ A\ B\ (abs\ s_1'\ s_2'\ A'\ B'\ M)\ N \longrightarrow_{beta} M\ N$$

**Figure 3** Definition of the theory $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$.

This theory needs of course to be completed case by case, so that $\widehat{\mathcal{S}}, \widehat{\mathcal{A}}, \widehat{\mathcal{R}}$ correctly represent $\mathcal{S}, \mathcal{A}, \mathcal{R}$. For this to hold, each sort $s \in \mathcal{S}$ should have a representation $\dot{s} : \widehat{\mathcal{S}}$, and this should restrict to a bijection when considering only the closed normal forms of type $\widehat{\mathcal{S}}$. Moreover, we should add rewrite rules such that $\mathcal{A}(s_1) = s_2$ iff $\widehat{\mathcal{A}}\ \dot{s}_1 \equiv \dot{s}_2$ and $\mathcal{R}(s_1, s_2) = s_3$ iff $\widehat{\mathcal{R}}\ \dot{s}_1\ \dot{s}_2 \equiv \dot{s}_3$.

In order to understand intuitively these conditions, let's look at an example.

▶ **Example 28.** The sort structure of Martin-Löf's Type Theory is given by the specification $\mathcal{S} = \mathbb{N}$, $\mathcal{A}(x) = x + 1$ and $\mathcal{R}(x, y) = max\{x, y\}$. We can represent this in Dedukti by declaring constants $z : \widehat{\mathcal{S}}$, $s[n : \widehat{\mathcal{S}}] : \widehat{\mathcal{S}}$ and rewrite rules $\widehat{\mathcal{A}}\ x \longrightarrow s\ x$, $\widehat{\mathcal{R}}\ z\ x \longrightarrow x$, $\widehat{\mathcal{R}}\ x\ z \longrightarrow x$ and $\widehat{\mathcal{R}}\ (s\ x)\ (s\ y) \longrightarrow s\ (\widehat{\mathcal{R}}\ x\ y)$.

Now one can proceed as before with the proofs of soundness, conservativity and adequacy, which follow the same idea as the previously presented ones. However, it is quite unsatisfying that we have to redo all the work of Sections 6 and 7 another time, and therefore one can wonder if we can reuse the results we already have about the first encoding.

Note that one may intuitively think of the $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$ as a "hidden implementation" of $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$. In this case, it should be possible to take a proof written in the $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$ and "implement" it in the $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$. Following this intuition, we define in the long version in [12] a notion of *theory morphism* which allows us to show the soundness of this new encoding by means of morphism from $(\Sigma_{\text{EPTS}}, \mathcal{R}_{\text{EPTS}})$ to $(\Sigma^{S}_{\text{EPTS}}, \mathcal{R}^{S}_{\text{EPTS}})$.

Nevertheless, this definition has too strong requirements and cannot be used to define a morphism in the other direction to show conservativity. Therefore, it is still an open problem for us to find a notion of morphism allowing to show the equivalence between the encodings. For the time being, in order to show conservativity (and then adequacy) of this new encoding one has to redo the work of Section 7.

## 9 The encoding in practice

Our encoding satisfies nice theoretical properties, but when using it in practice it becomes quite annoying to have to explicit all the information needed in $app_{s_1,s_2}$ and $abs_{s_1,s_2}$. Worst, when performing translations from other systems where those parameters are not explicit we would then have to compute them during the translation. Thankfully, LAMBDAPI[10], an implementation of DEDUKTI, allows us to solve this by declaring some arguments as implicit, so they are only calculated internally.

Using the encoding of Figure 3 we can mark for instance the arguments $s_1, s_2, A$ of $Prod$ as implicit. We can then also rename $Prod$ into $\Pi'$, $abs$ into $\lambda'$, $app$ into $\blacksquare$ and use another LAMBDAPI feature allowing to mark $\Pi', \lambda'$ as quantifier and $\blacksquare$ as infix left. This then allows us to represent $\Pi x : A.B$ as $\Pi' x : El\ [\![A]\!].[\![B]\!]$, $\lambda x : A.B$ as $\lambda' x : El\ [\![A]\!].[\![B]\!]$ and $M\ N$ as $[\![M]\!] \blacksquare [\![N]\!]$. Using these notations, we can write terms in the encoding in a natural way, and we refer to `https://github.com/thiagofelicissimo/examples-encodigs` for a set of examples of this.

However, as DEDUKTI also aims to be used in practice for sharing real libraries between proof assistants, we also tested how our approach copes with more practical scenarios. We provide in `https://github.com/thiagofelicissimo/encoding-benchmarking` a benchmark of Fermat's little theorem library in DEDUKTI[22], where we compare the traditional encoding with an adequate version that applies the ideas of our approach[5]. As we can see, the move from the traditional to the adequate version introduces a considerable performance hit. The standard DEDUKTI implementation, which is our reference here, takes 16 times more time to typecheck the files. This is probably caused by the insertion of type parameters $A$ and $B$ in $abs_{s_A,s_B}$ and $app_{s_A,s_B}$, which are not needed in traditional encodings.

Nevertheless, DEDUKTI is still able to typecheck our encoding within reasonable time, showing that our approach is indeed usable in practical scenarios, even if it is not the most performing one. Moreover, as our encoding is mainly intended to be used to check proofs, and not with interactive proof development, immediacy of the result is not essential and thus it can be reasonable to trade performance for better theoretical properties. Still, we plan in the future to look at techniques to improve our performances. In particular, using more sharing in DEDUKTI would probably reduce the time for typechecking, as the parameter annotations in $app_{s_A,s_B}$ and $abs_{s_A,s_B}$ carry a lot of repetition.

## 10 Conclusion

By separating the framework's abstraction and application from the ones of the encoded system, we have proposed a new paradigm for DEDUKTI encodings. Our approach offers much more well-behaved encodings, whose conservativity can be shown in a much more straightforward way and which feature adequacy theorems, something that was missing from traditional DEDUKTI encodings. However, differently from the LF approach, our encoding is also computational. Therefore, our method combines the adequacy of LF encodings with the computational aspect of DEDUKTI encodings.

By decoupling the framework's $\beta$ from the rewriting of the encoded system, our approach allows to show the expected properties of the encoding without requiring to show that the encoded system terminates. Indeed, our adequacy result concerns all functional EPTS, even non terminating ones, such as the one with $Type : Type$. This sets our work apart from [9], whose conservativity proof requires the encoded system to be normalizing.

[5] Because the underlying logic of the library is not a PTS, this encoding is not exactly the one we present here. However, it uses the same ideas discussed, and the same proof strategy to show adequacy applies.

This work opens many other directions we would like to explore. We believe that our technique can be extended to craft adequate and computational encodings of type theories with much more complex features, such as (co)inductive types, universe polymorphism, predicate subtyping and others. For instance, in the case of inductive types no type-level rewriting rules need to be added, thus *$\beta$ is SN in Dedukti* (Theorem 5) would apply. Therefore, we could repeat the same technique of normalizing only with $\beta$ to show conservativity.

However, we would be particularly interested to see if we could take a general definition of type theories covering most of these features (maybe in the lines of [5]). This would allow us to define a single encoding which could be applied to encode various features, and thus would saves us from redoing similar proofs multiple times.

## References


1. Ali Assaf. *A framework for defining computational higher-order logics.* Thesis, École polytechnique, September 2015. URL: https://pastel.archives-ouvertes.fr/tel-01235303.
2. Ali Assaf. Conservativity of embeddings in the lambda pi calculus modulo rewriting. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
3. Ali Assaf, Guillaume Burel, Raphaël 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. Gilles Barthe. The relevance of proof-irrelevance. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 755–768, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
5. Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories, 2020. arXiv:2009.05539.
6. Frédéric Blanqui. *Théorie des types et réécriture. (Type theory and rewriting).* PhD thesis, University of Paris-Sud, Orsay, France, 2001. URL: https://tel.archives-ouvertes.fr/tel-00105522.
7. Frédéric Blanqui. Type safety of rewrite rules in dependent types. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*, volume 167, page 14, Paris, France, June 2020. doi:10.4230/LIPIcs.FSCD.2020.13.
8. Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. 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 9:1–9:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.9.
9. Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
10. Deducteam. Lambdapi. https://github.com/Deducteam/lambdapi.
11. Gilles Dowek. Models and termination of proof reduction in the lambda pi-calculus modulo theory. In *ICALP*, 2017.
12. Thiago Felicissimo. Adequate and computational encodings in the logical framework Dedukti. Draft at https://lmf.cnrs.fr/Perso/ThiagoFelicissimo, 2022.
13. Thiago Felicissimo. No need to be implicit! Draft at https://lmf.cnrs.fr/Perso/ThiagoFelicissimo, 2022.
14. Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework.* Thesis, Université Paris-Saclay, June 2021. URL: https://tel.archives-ouvertes.fr/tel-03418761.

15 Guillaume Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, Paris-Saclay, 2020. Thèse de doctorat dirigée par Blanqui, Frédéric et Hermant, Olivier. URL: `http://www.theses.fr/2020UPASG045`.

16 Robert Harper. An equational logical framework for type theories. *arXiv preprint*, 2021. `arXiv:2106.01484`.

17 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. `doi:10.1145/138027.138060`.

18 Gabriel Hondet and Frédéric Blanqui. Encoding of Predicate Subtyping with Proof Irrelevance in the λΠ-Calculus Modulo Theory. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.TYPES.2020.6`.

19 Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1):279–308, 1993. `doi:10.1016/0304-3975(93)90091-7`.

20 Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs*, pages 254–276, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

21 Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22:153–180, 2012.

22 François 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. `doi:10.4204/EPTCS.274.5`.

23 François Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.


## A Typing rules and metatheory of Dedukti

▶ **Proposition 29** (Basic properties). *Suppose $\longrightarrow_{\beta\mathcal{R}}$ is confluent.*

1. *Weakening: If* $\Sigma;\Gamma \vdash M : A$, $\Gamma \sqsubseteq \Gamma'$ *and* $\Sigma;\Gamma'$ `well-formed` *then* $\Sigma;\Gamma' \vdash M : A$
2. *Well-typedeness of contexts: If* $\Sigma;\Gamma$ `well-formed` *then for all* $x : B \in \Gamma$, $\Sigma;\Gamma \vdash B : \mathtt{TYPE}$
3. *Inversion of typing: Suppose* $\Sigma;\Gamma \vdash M : A$
   - *If* $M = x$ *then* $x : A' \in \Gamma$ *and* $A \equiv A'$
   - *If* $M = c[\vec{N}]$ *then* $c[\Delta] : A' \in \Sigma$, $\Sigma;\Delta \vdash A' : s$, $\Sigma;\Gamma \vdash \vec{N} : \Delta$ *and* $A'\{\vec{N}/\Delta\} \equiv A$
   - *If* $M = \mathtt{TYPE}$ *then* $A \equiv \mathtt{KIND}$
   - $M = \mathtt{KIND}$ *is impossible*
   - *If* $M = \Pi x : A_1.A_2$ *then* $\Sigma;\Gamma \vdash A_1 : \mathtt{TYPE}$, $\Sigma;\Gamma, x : A_1 \vdash A_2 : s$ *and* $s \equiv A$
   - *If* $M = M_1 M_2$ *then* $\Sigma;\Gamma \vdash M_1 : \Pi x : A_1.A_2$, $\Sigma;\Gamma \vdash M_2 : A_1$ *and* $A_2\{M_2/x\} \equiv A$
   - *If* $M = \lambda x : B.N$ *then* $\Sigma;\Gamma \vdash B : \mathtt{TYPE}$, $\Sigma;\Gamma, x : B \vdash C : s$, $\Sigma;\Gamma, x : B \vdash N : C$ *and* $A \equiv \Pi x : B.C$
4. *Uniqueness of types: If* $\Sigma;\Gamma \vdash M : A$ *and* $\Sigma;\Gamma \vdash M : A'$ *then* $A \equiv A'$
5. *Well-sortness: If* $\Sigma;\Gamma \vdash M : A$ *then* $\Sigma;\Gamma \vdash A : s$ *or* $A = \mathtt{KIND}$

▶ **Theorem 30** (Conv in context for DK). *Let* $A \equiv A'$ *with* $\Sigma;\Gamma \vdash A' : s$. *We have*

- $\Sigma;\Gamma, x : A, \Gamma'$ `well-formed` $\Rightarrow \Sigma;\Gamma, x : A', \Gamma'$ `well-formed`
- $\Sigma;\Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Sigma;\Gamma, x : A', \Gamma' \vdash M : B$

▶ **Proposition 31** (Reduce type in judgement). *Suppose $\longrightarrow_{\beta\mathcal{R}}$ is confluent and satisfies subject reduction. Then if* $\Sigma;\Gamma \vdash M : A$ *and* $A \longrightarrow^* A'$ *we have* $\Sigma;\Gamma \vdash M : A'$.

# B Metatheory of Explicitly-typed Pure Type Systems

We have the following properties for functional EPTSs. We refer to [13] for the proofs.

▶ **Proposition 32** (Weakening). *Let* $\Gamma \sqsubseteq \Gamma'$ *with* $\Gamma'$ well-formed. *If* $\Gamma \vdash M : A$ *then* $\Gamma' \vdash M : A$.

▶ **Proposition 33** (Inversion). *If* $\Gamma \vdash M : C$ *then*

- *If* $M = x$, *then*
  - $\Gamma$ well-formed *with a smaller derivation tree*
  - *there is* $x$ *with* $x : A \in \Gamma$ *and* $C \equiv A$
- *If* $M = s$, *then there is* $s'$ *with* $(s, s') \in \mathcal{A}$ *and* $C \equiv s'$
- *If* $M = \Pi_{s_1,s_2}(A, [x]B)$ *then*
  - $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
  - $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
  - *there is* $s_3$ *with* $(s_1, s_2, s_3) \in \mathcal{R}$ *and* $C \equiv s_3$
- *If* $M = \lambda_{s_1,s_2}(A, [x]B, [x]N)$ *then*
  - $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
  - $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
  - *there is* $s_3$ *with* $(s_1, s_2, s_3) \in \mathcal{R}$
  - $\Gamma, x : A \vdash N : B$ *with a smaller derivation tree*
  - $C \equiv \Pi_{s_1,s_2}(A, [x]B)$
- *If* $M = @_{s_1,s_2}(A, [x]B, N_1, N_2)$ *then*
  - $\Gamma \vdash A : s_1$ *with a smaller derivation tree*
  - $\Gamma, x : A \vdash B : s_2$ *with a smaller derivation tree*
  - *there is* $s_3$ *with* $(s_1, s_2, s_3) \in \mathcal{R}$
  - $\Gamma \vdash N_1 : A$ *with a smaller derivation tree*
  - $\Gamma \vdash N_2 : \Pi_{s_1,s_2}(A, [x]B)$ *with a smaller derivation tree*
  - $C \equiv B\{N_2/x\}$

▶ **Proposition 34** (Uniqueness of types). *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *we have* $A \equiv B$.

▶ **Corollary 35** (Uniqueness of sorts). *If* $\Gamma \vdash M : s$ *and* $\Gamma \vdash M : s'$ *we have* $s = s'$.

▶ **Proposition 36** (Conv in context). *Let* $A \equiv A'$ *and* $\Gamma \vdash A' : s$. *We have*

- $\Gamma, x : A, \Gamma'$ well-formed $\Rightarrow \Gamma, x : A', \Gamma'$ well-formed
- $\Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Gamma, x : A', \Gamma' \vdash M : B$

▶ **Proposition 37** (Substitution in judgment). *Let* $\Gamma \vdash N : A$. *We have*

- $\Gamma, x : A, \Gamma'$ well-formed $\Rightarrow \Gamma, \Gamma'\{N/x\}$ well-formed
- $\Gamma, x : A, \Gamma' \vdash M : B \Rightarrow \Gamma, \Gamma'\{N/x\} \vdash M\{N/x\} : B\{N/x\}$

# mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity

**Clément Aubert**
School of Computer and Cyber Sciences, Augusta University, GA, USA

**Thomas Rubiano**
LIPN – UMR 7030 Université Sorbonne Paris Nord, France

**Neea Rusch**
School of Computer and Cyber Sciences, Augusta University, GA, USA

**Thomas Seiller**
LIPN – UMR 7030 Université Sorbonne Paris Nord, France
CNRS, Paris, France

## Abstract

Implicit Computational Complexity (ICC) drives better understanding of complexity classes, but it also guides the development of resources-aware languages and static source code analyzers. Among the methods developed, the *mwp-flow analysis* [23] certifies polynomial bounds on the size of the values manipulated by an imperative program. This result is obtained by bounding the transitions between states instead of focusing on states in isolation, as most static analyzers do, and is not concerned with termination or tight bounds on values. Those differences, along with its built-in compositionality, make the mwp-flow analysis a good target for determining how ICC-inspired techniques diverge compared with more traditional static analysis methods. This paper's contributions are three-fold: we fine-tune the internal machinery of the original analysis to make it tractable in practice; we extend the analysis to function calls and leverage its machinery to compute the result of the analysis efficiently; and we implement the resulting analysis as a lightweight tool to automatically perform data-size analysis of `C` programs. This documented effort prepares and enables the development of certified complexity analysis, by transforming a costly analysis into a tractable program, that furthermore decorrelates the problem of deciding if a bound exist with the problem of computing it.

**2012 ACM Subject Classification** Software and its engineering → Automated static analysis; Theory of computation → Complexity theory and logic; Theory of computation → Logic and verification

**Keywords and phrases** Static Program Analysis, Implicit Computational Complexity, Automatic Complexity Analysis, Program Verification

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.26

**Related Version** *Technical report*: `https://hal.archives-ouvertes.fr/hal-03596285` [4]

**Supplementary Material** *Software (Source Code)*: `https://github.com/statycc/pymwp`
archived at `swh:1:dir:22a4ab0cfad49138981ed25fc2abfe830fb7ccdf`
*Software (Documentation and Demo)*: `https://statycc.github.io/pymwp`

**Funding** This research is supported by the Th. Jefferson Fund of the Embassy of France in the United States and the FACE Foundation, and has benefited from the research meeting 21453 "Static Analyses of Program Flows: Types and Certificate for Complexity" in Schloss Dagstuhl. Th. Rubiano and Th. Seiller are supported by the Île-de-France region through the DIM RFSI project "CoHOp".

**Acknowledgements** The authors wish to express their gratitude to Assya Sellak for her contribution to this work, to the reviewers of previous versions for their comments, and to the FSCD community: in particular, the reviews we received were extremely interesting, and generated new directions and questions, for which we are thankful.

© Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller;
licensed under Creative Commons License CC-BY 4.0

# 1 Introduction: letting ICC drive the development of static analyzers

Certifying program resource usages is possibly as crucial as the specification of program correctness, since a guaranteed correct program whose memory usage exceeds available resources is, in fact, unreliable. The field of Implicit Computational Complexity (ICC) theory [15] pioneers in "embedding" in the program itself a guarantee of its resource usage, using e.g., bounded recursion [8, 27] or type systems [6, 26]. This field initiated numerous distinct and original approaches, primarily to characterize complexity classes in a machine-independent way, with increasing expressivity, but these approaches have rarely materialized into concrete programming languages or program analyzers: even if, as opposed to traditional complexity, its models are generally expressive enough to write down actual algorithms [30, p. 11], they rarely escape the sphere of academia or extend beyond toy languages, with a few exceptions [5, 22]. However, by abstracting away constant factors and insignificant orders of magnitude, it is frequently conjectured that ICC will allow sidestepping some of the difficult issues one usually has to face when inferring the resource usage of a concrete program.

This work reinforces this conjecture by adjusting, improving and implementing an existing ICC technique, the *mwp-bounds analysis* [23], which certifies that the values computed by an imperative program will be bounded by polynomials in the program's input. This flow analysis is elegant but computationally costly, and it missed an opportunity to leverage its built-in compositionality: we address both issues by revisiting and expanding the original flow calculus, and further make our point by implementing it on a subset of the `C` programming language. While the theory has been improved to allow analysis of function definitions and calls – including recursive ones, a feature not widely supported [21, p. 359] – , its integration into the implementation is underway, as we placed primary focus on developing an efficient and implementable technique for program analysis. Implementing a tool along the theory enabled testing improvements in real-life, which in return drove adjustments to the theory.

Our enhanced technique answers positively two questions asked by the authors of the original analysis [23, Section 1.2], namely

1. Can the method be extended to richer languages?
2. Can it lead to powerful and convenient tools?

It also supports the conjecture that ICC can be used to construct concrete tools, but highlights that doing so requires adjusting the theory to make it tractable in practice. This work also provides better insight into the original analysis, by e.g., separating the algorithm to decide the existence of a bound from its evaluation into a concrete bound; and by illustrating its plasticity: while our analysis conservatively extends the original one, it nevertheless greatly alters its internal machinery to ease its implementability. Last but not least, our technique is orthogonal to most static analysis methods, which focus on worst-case resource-usage complexity or termination, while ours establishes that the growth rate of variables values is at most polynomially related to their inputs.

Our paper starts by recalling the "original" *mwp-bounds analysis* [23] – to which we refer for a more gentle introduction – and discuss its limitations (Sect. 2). In Sect. 3, we motivate, introduce and justify two modifications to this original analysis, and state that this calculus can be reduced to the original one. We then extend this analysis along two axis (Sect. 4): we detail how functions calls can be analyzed, and how the structures we implemented allowed to speed up some very costly operations. Finally, Sect. 5 presents and discuss our implementation, and Sect. 6 concludes. The proofs, some additional details on semi-rings and the detail of our benchmarks are in appendix, with the exception of some tedious proofs relative to semi-rings that are only in our technical report [4].

# 2 Background: the original flow analysis

The original analysis [23] computes a polynomial bound – if it exists – on the sizes (of the value itself) of variables in an imperative `while` programming language, extended with a `loop` operator, by computing for each variable a vector that tracks how it depends on other variables – and the program itself gets assigned a matrix collecting those vectors. While this does not ensure termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that *if* it terminates, it will do so in polynomial time.

## 2.1 Language analyzed: fragments of imperative language

▶ **Definition 1** (Imperative Language). *Letting natural number variables range over* `X` *and* `Y` *and boolean expressions over* `b`*, we define* expressions `e` *and* commands `C` *as follows:*

```
e ::= X || X - Y || X + Y || X * Y
C ::= X = e || if b then C else C || while b do {C} || loop X {C} || C ; C
```

*where* `loop X {C}` *means "do* `C` `X` *times" and* `C;C` *is used for sequentiality ("do* `C`*, then* `C`*"). We write "program" for a series of commands composed sequentially.*

This language assumes that the program's inputs are the only variables, and that assigning a value to a variable inside the program is not permitted. Extending flow calculi to those operations has been discussed [23, p. 3] and proven possible [9], but we leave this for future work – in particular, our `C` examples will be of `foo` functions with their variables listed as parameters[1]. However, we disallow w.l.o.g. composed expressions of the form `X + Y * Y`, which can always be dealt with in the style of three-address code.

## 2.2 A flow calculus of mwp-bounds for complexity analysis

*Flows* characterize controls from one variable to another, and can be, in increasing growth rate, of type 0 – the absence of any dependency – *m*aximum, *w*eak polynomial and *p*olynomial. The bounds on programs written in the syntax of Sect. 2.1 are represented and calculated thanks to vectors and matrices whose coefficients are elements of the mwp semi-ring.

▶ **Definition 2** (The mwp semi-ring and matrices over it). *Letting* $\textsc{mwp} = \{0, m, w, p\}$ *with* $0 < m < w < p$*, and* $\alpha$*,* $\beta$*,* $\gamma$ *range over* $\textsc{mwp}$*, the* mwp semi-ring $(\textsc{mwp}, 0, m, +, \times)$ *is defined with* $+ = \max$*,* $\alpha \times \beta = \max(\alpha, \beta)$ *if* $\alpha, \beta \neq 0$*, and* $0$ *otherwise.*

*We denote* $\mathbb{M}(\textsc{mwp})$ *the matrices over* $\textsc{mwp}$*, and, fixing* $n \in \mathbb{N}$*,* $M$ *for* $n \times n$ *matrices over* $\textsc{mwp}$*,* $M_{ij}$ *for the coefficient in the* $i$*th row and* $j$*th column of* $M$*,* $\oplus$ *for the componentwise addition, and* $\otimes$ *for the product of matrices defined in a standard way. The* $0$*-element for addition is* $0_{ij} = 0$ *for all* $i, j$*, and the* $1$*-element for product is* $1_{ii} = m$*,* $1_{ij} = 0$ *if* $i \neq j$*, and the resulting structure* $(\mathbb{M}(\textsc{mwp}), 0, 1, \otimes, \oplus)$ *is a semi-ring that we simply write* $\mathbb{M}(\textsc{mwp})$*. The closure operator* $\cdot^*$ *is* $M^* = 1 \oplus M \oplus (M^2) \oplus \ldots$*, for* $M^0 = 1$*,* $M^{m+1} = M \otimes M^m$*.*

[1] Our implementation allows to relax this condition, as exemplified in `inline_variable.c`, without losing any of the results expressed in this paper. Assuming a fixed number of variables, known ahead of time, is mostly a theoretical artifact used to simplify the analysis.

$$\frac{}{\vdash_{\text{JK}} \texttt{Xi} : \{^m_{\texttt{i}}\}} \text{E1} \qquad \frac{}{\vdash_{\text{JK}} \texttt{e} : \{^w_{\texttt{i}} \mid \texttt{Xi} \in \text{var}(\texttt{e})\}} \text{E2}$$

$$\star \in \{+,-\} \frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2}{\vdash_{\text{JK}} \texttt{Xi}\star\texttt{Xj} : pV_1 \oplus V_2} \text{E3} \qquad \star \in \{+,-\} \frac{\vdash_{\text{JK}} \texttt{Xi} : V_1 \quad \vdash_{\text{JK}} \texttt{Xj} : V_2}{\vdash_{\text{JK}} \texttt{Xi}\star\texttt{Xj} : V_1 \oplus pV_2} \text{E4}$$

**(a)** Rules for assigning vectors to expressions.

$$\frac{\vdash_{\text{JK}} \texttt{e} : V}{\vdash_{\text{JK}} \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V} \text{A} \qquad \frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{C1; C2} : M_1 \otimes M_2} \text{C}$$

$$\frac{\vdash_{\text{JK}} \texttt{C1} : M_1 \quad \vdash_{\text{JK}} \texttt{C2} : M_2}{\vdash_{\text{JK}} \texttt{if b then C1 else C2} : M_1 \oplus M_2} \text{I}$$

$$\forall i, M^*_{ii} = m \ \frac{\vdash_{\text{JK}} \texttt{C} : M}{\vdash_{\text{JK}} \texttt{loop Xl \{C\}} : M^* \oplus \{^p_{\texttt{l}} \to j \mid \exists i, M^*_{ij} = p\}} \text{L}$$

$$\forall i, M^*_{ii} = m \text{ and } \forall i, j, M^*_{ij} \neq p \ \frac{\vdash_{\text{JK}} \texttt{C} : M}{\vdash_{\text{JK}} \texttt{while b do \{C\}} : M^*} \text{W}$$

**(b)** Rules for assigning matrices to commands.

**Figure 1** Original non-deterministic ("Jones-Kristiansen") flow analysis rules.

Although not crucial to understand our development, details about (strong) semi-rings and the mwp semi-ring, and the construction of a semi-ring whose elements are matrices with coefficients in a semi-ring – so, in particular, $\mathbb{M}(\text{MWP})$ – are given in our technical report [4, A.1 and A.2] and sketched in appendix Appendix A.

Below, we let $V_1$, $V_2$ be column vectors with values in MWP, $\alpha V_1$ be the usual scalar product, and $V_1 \oplus V_2$ be defined componentwise. We write $\{^\alpha_i\}$ for the vector with 0 everywhere except for $\alpha$ in its $i$th row, and $\{^\alpha_i, ^\beta_j\}$ for $\{^\alpha_i\} \oplus \{^\beta_j\}$.

Replacing in a matrix $M$ the $j$th column vector by $V$ is denoted $M \overset{j}{\leftarrow} V$. The matrix $M$ with $M_{ij} = \alpha$ and 0 everywhere else is written $\{^\alpha_i \to j\}$, and the set of variables in the expression `e` is written var(`e`). The assumption is made that exactly $n$ different variables are manipulated throughout the analyzed program, so that $n$-vectors are assigned to expressions – in a non-deterministic way, to capture larger classes of programs [23, Section 8] – and $n \times n$ matrices are assigned to commands using the rules presented Fig. 1 [23, Section 5].

The intuition is that if $\vdash_{\text{JK}}$ `C` : $M$ can be derived, then all the values computed by `C` will grow at most polynomially w.r.t. its inputs [23, Theorem 5.3], e.g., will be bounded by $\max(\vec{x}, p_1(\vec{y})) + p_2(\vec{z})$, where $p_1$ and $p_2$ are polynomials and $\vec{x}$ (resp. $\vec{y}$, $\vec{z}$) are $m$-(resp. $w$-, $p$-)annotated variables in the vector for the considered output. Since the derivation system is non-deterministic, multiple matrices and polynomial bounds – that sometimes coincide – may be assigned to the same program. Furthermore, the coefficient at $M_{\texttt{ij}}$ carries quantitative information about the way `Xi` depends on `Xj`, knowing that 0- and $m$-flows are harmless and without constraints, but that $w$- and $p$- flows are more harmful w.r.t. polynomial bounds and need to be handled with care, particularly in loops – hence the condition on the L and W rules. The derivation may fail – some programs may not be assigned a matrix – if at least one of the variables used in the body of a loop depends "too strongly" upon another, making it impossible to ensure polynomial bounds on the loop itself. We will use the following example as a common basis to discuss possible failure, non-determinism, and our improvements.

▶ **Example 3.** Consider `loop X3 {X2 = X1 + X2}`. The body of the `loop` command admits three different derivations, obtained by applying A to one of the three derivation of the expression `X1 + X2`, that we name $\pi_0$, $\pi_1$ and $\pi_2$:

$$\dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\text{E1} \quad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}}\text{E3} \qquad \dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\text{E1} \quad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}\text{E4} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\text{E2}$$

From $\pi_0$, the derivation of `loop X3 {X2 = X1 + X2}` can be completed using A and L, but since L requires having only $m$ coefficients on the diagonal, $\pi_1$ cannot be used to complete the derivation, because of the $p$ coefficient in a box below:

$$\dfrac{\dfrac{\begin{matrix} \vdots\, \pi_0 \\ \vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix} \end{matrix}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}\text{A}}{\vdash_{\text{JK}} \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix}}\text{L} \qquad \dfrac{\begin{matrix} \vdots\, \pi_1 \\ \vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix} \end{matrix}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & \boxed{p} & 0 \\ 0 & 0 & m \end{pmatrix}}\text{A}$$

Similarly, using A after $\pi_2$ gives a $w$ coefficient on the diagonal and makes it impossible to use L, hence only one derivation for this program exists.

## 2.3 Limitations and inefficiencies of the mwp analysis

Even if the proof techniques are far from trivial, with only 9 rules and skipping over boolean expressions (observe that the condition `b` has no impact in the rules I or W), the analysis is flexible and easy to carry out – at least mathematically. It also has inherent limitations: while the technique is sound, it is not complete and programs such as greatest common divisor fail to be assigned a matrix. We will discuss in Sect. 5.2, the benefits and originality of this analysis, but we would now like to stress how it is computationally inefficient, since the non-determinacy makes the analysis costly to carry out and can lead to memory explosions.

Abstracting Example 3, one can see that the base case of non-determinism – e.g., to assign a vector to `X1 ⋆ X2`–yields vectors $\binom{p}{m}$ (using E1 then E3), $\binom{m}{p}$ (using E1 then E4) and $\binom{w}{w}$ (using E2). Since none of those vectors is less than the others, only two strategies are available to analyze a larger program containing `X1 ⋆ X2`: either the derivations for this base case are considered one after the other, or they are all stored in memory at the same time. Considering the derivations for the base case one after the other can lead to a time explosion, as a program of $n$ lines can have $3^n$ different derivations – as exemplified by `explosion.c`, a simple series of applications – and it is possible that only one of them can be completed, so all must be explored. On the other hand, storing those three vectors and constructing all the matrices in parallel leads to a memory explosion: the analysis for two commands involving 6 variables, with 3 choices – which cannot be simplified as explained previously – would result in 9 matrices of size $6 \times 6$, i.e., 324 coefficients. All in all, a program of $n$ lines with $x$ different variables can require $c_1^n$ different derivations, which can produce up to $(c_2 \times x)^2$ coefficients to store for some constants $c_1$, $c_2$.

Beyond inefficiency, there are additional limitations: while the analysis is naturally compositional, this feature is not leveraged in the original system; furthermore, an occurrence of non-polynomial flows in the matrix causes the analysis to simply stop, thus not capturing failure in a meaningful way. We will discuss our solutions to these deficiencies next.

$$\frac{\star \in \{+,-\}}{\vdash \texttt{Xi}\star\texttt{Xj} : (0 \mapsto \{{}^{m}_{\texttt{i}}, {}^{p}_{\texttt{j}}\}) \oplus (1 \mapsto \{{}^{p}_{\texttt{i}}, {}^{m}_{\texttt{j}}\}) \oplus (2 \mapsto \{{}^{w}_{\texttt{i}}, {}^{w}_{\texttt{j}}\})}\ \mathrm{E}^{\mathrm{A}}$$

$$\frac{}{\vdash \texttt{Xi * Xj} : \{{}^{w}_{\texttt{i}}, {}^{w}_{\texttt{j}}\}}\ \mathrm{E}^{\mathrm{M}} \qquad \frac{}{\vdash \texttt{Xi} : \{{}^{m}_{\texttt{i}}\}}\ \mathrm{E}^{\mathrm{S}}$$

**(a)** Rules for assigning vectors to expressions.

$$\frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \mathrm{A} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{C1; C2} : M_1 \otimes M_2}\ \mathrm{C} \qquad \frac{\vdash \texttt{C1} : M_1 \quad \vdash \texttt{C2} : M_2}{\vdash \texttt{if b then C1 else C2} : M_1 \oplus M_2}\ \mathrm{I}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{loop Xl \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{p}_{\texttt{l}} \to j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}^{\infty}$$

$$\frac{\vdash \texttt{C} : M}{\vdash \texttt{while b do \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{\infty}_{i} \to j \mid M^*_{ij} = p\}}\ \mathrm{W}^{\infty}$$

**(b)** Rules for assigning matrices to commands.

**Figure 2** Deterministic improved flow analysis rules.

## 3 A deterministic, always-terminating, declension of the mwp analysis

The problem of finding a derivation in the original calculus is in NP [23, Theorem 8.1]. But since all the non-determinism is in the rules to assigning a vector, the potentially exponential number of derivations are actually extremely similar. Hence, instead of having the analysis stop when failing to establish a derivation and re-starting from scratch, storing the different vectors and constructing the derivation while keeping all the options open seems to be a better strategy, but, as we have seen, this causes a memory blow-up. We address it by fine-tuning the internal machinery: to represent non-determinism, we let the matrices take as values either functions from choices to coefficients in MWP or coefficients in MWP, so that instead of mapping choices to derivations, all the derivations are represented by the same matrix that internalizes the different choices. Sect. 3.1 discusses this improvement, which results in a notable gain: getting back to the example of Sect. 2.3, a program involving 6 variables, with 3 choices, would now be assigned a (unique) $6 \times 6$ matrix that requires 66 coefficients instead of the 324 we previously had – this is because 30 coefficients are "simple" values in MWP, and 6 are functions from a set of choices $\{0, 1, 2\}$ to values in MWP, each represented with 6 coefficients.

For the choices that give coefficients fulfilling the side condition of L or W, the derivation can proceed as usual, but when a particular choice gives a coefficient that violates it, we decided against simply removing it. Instead, to guarantee that all derivations always terminate, we mark that choice by indicating that it would not provide a polynomial bound. This requires extending the MWP semi-ring with a special value $\infty$ that represents failure in a local way, marking non-polynomial flows, and is detailed in Sect. 3.2. As a by-product, this enables fine-grained information on programs that *do not* have polynomially bounded growth, since the precise dependencies that break this growth rate can be localized.

Taken together (Sect. 3.3), our improvements ensure that exactly one matrix will always be assigned to a program while carrying over the correctness of the original analysis. We give in Fig. 2 the deterministic system we are introducing in full, but will gently introduce it though the remaining parts of this section: note that the rules A, C and I are unchanged, up to the fact that the matrices, sum and product are in a different semi-ring.

## 3.1 Internalizing non-determinism: the choice data flow semi-rings

Internalizing the choice requires altering the semi-ring used in the analysis: we want to replace the three vectors over MWP that can be assigned to an expression by a single vector over $\{0,1,2\} \to$ MWP that captures the same three choices. For a program needing to decide $p$ times between the 3 available choices, this means replacing the $3 \times p$ different matrices in $\mathbb{M}(\text{MWP})$ by a single matrix in $\mathbb{M}(\{0,1,2\}^p \to \text{MWP})$. For any strong semi-ring $\mathbb{S}$ and family of sets $(A_i)_{i=1,\dots,p}$, both $A_i \to \mathbb{S}$ and $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S})$ are semi-rings, using the usual cartesian product of sets, and there exists an isomorphism $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S}) \cong \prod_{i=1}^p A_i \to \mathbb{M}(\mathbb{S})$ [4, A.3]. This dual nature of the semi-ring considered is useful:

- the analysis will now assign an element $M$ of $\mathbb{M}(\prod_{i=1}^p A_i \to \text{MWP})$ to a program;
- representing $M$ as an element of $\prod_{i=1}^p A_i \to \mathbb{M}(\text{MWP})$ allows one to use an *assignment* $\vec{a} = (a_1, \dots, a_p) \in \prod_{i=1}^p A_i$ to produce a matrix $M[\vec{a}] \in \mathbb{M}(\text{MWP})$, recovering the mwp-flow that would have been computed by making the choices $a_1, \dots, a_p$ in the derivation.

▶ Remark 4. As the unique degree of non-determinism to assign a matrix to commands is 3, our modification of the analysis flow consists simply of recording the different choices by letting $A_i = \{0,1,2\}$ for all $i = 1, \dots, p$ where $p$ is the number of times a choice had to be taken. Starting with Sect. 4, function calls will require potentially different sets $A_i$.

▶ Notation 5. In the following and in the implementation alike, we will denote a function $(a_1^0 \times \cdots \times a_p^0 \mapsto \alpha_0) + \cdots + (a_1^k \times \cdots \times a_p^k \mapsto \alpha_k)$ in $A^p \to$ MWP with $\text{Card}(A) = k$ by, omitting the product, $(\alpha_0 \delta(a_1^0, 0) \cdots \delta(a_p^0, p)) + \cdots + (\alpha_k \delta(a_1^k, 0) \cdots \delta(a_p^k, p))$, with $\delta(i,j) = m$ if the $j$th choice is $i$, 0 otherwise. Example 8 will justify and explain this choice.

Our derivation system replaces the E3 and E4 rules with a single rule $\text{E}^{\text{A}}$ ("additive"), and splits E2 in two exclusive rules, $\text{E}^{\text{M}}$ for "multiplicative" and $\text{E}^{\text{S}}$ for "simple" (atomic) expressions – Theorem 11 will prove how they are equivalent.

▶ **Example 6.** We represent the vectors $\begin{pmatrix} p \\ m \\ 0 \end{pmatrix}$, $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$ from Example 3 with a single vector $\begin{pmatrix} p\delta(0,0)+m\delta(1,0)+w\delta(2,0) \\ m\delta(0,0)+p\delta(1,0)+w\delta(2,0) \\ 0 \end{pmatrix}$, that can be read as $\begin{pmatrix} \{0\mapsto p, 1\mapsto m, 2\mapsto w\} \\ \{0\mapsto m, 1\mapsto p, 2\mapsto w\} \\ 0 \end{pmatrix}$, where we write 0 for $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0\}$[2]. Since in particular[3], $\mathbb{M}(\{0,1,2\} \to \text{MWP}) \cong \{0,1,2\} \to \mathbb{M}(\text{MWP})$, the obtained vector can be rewritten as $0 \mapsto \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}, 1 \mapsto \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}, 2 \mapsto \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

## 3.2 Internalizing failure: de-correlating derivations and bounds

The original analysis stops when detecting a non-polynomial flow, puts an end to the chosen strategy (i.e., set of choices) and restarts from scratch with another one. We adapt the rules so that every derivation can be completed even in the presence of non-polynomial flows, thanks to a new top element, $\infty$, representing failure in a local way.

Ignoring our previous modification in this subsection, the semi-ring $\text{MWP}^\infty$ we need to consider is $(\text{MWP} \cup \{\infty\}, 0, m, +^\infty, \times^\infty)$, with $\infty > \alpha$ for all $\alpha \in$ MWP, $+^\infty = \max$ as before, and $\alpha \times^\infty \beta = 0$ if $\alpha, \beta \neq \infty$ and $\alpha$ or $\beta$ is 0, $\max(\alpha, \beta)$ otherwise. This different condition in the definition of $\times^\infty$ ensures that once non-polynomial flows have been detected, they cannot be erased (as $\infty \times^\infty 0 = \infty$).

[2] The implementation supports both coefficients from MWP *and* coefficients from $\{0,1,2\}^p \to$ MWP, cf. e.g., a simple assignment example `assign_expression.c`.

[3] This is a variant of Lemma 21 [4, A.3]. While the latter lemma applies to algebras of square matrices, a similar result holds for rectangular matrices of a fixed size; the algebraic structure is no longer that of a semi-ring as rectangular matrices do not possess a proper multiplication, but the proof can be adapted to show the existence of an isomorphism of modules between the considered spaces.

The only cases where the original analysis may fail is if the side conditions of L or W (Fig. 1) are not met. We replace those by $L^\infty$ and $W^\infty$ (Fig. 2), which replace the problematic coefficients with $\infty$, marking non-polynomial dependencies, and carry on the analysis.

▶ **Example 7.** The program from Example 3 would now receive three derivations (omitting the one obtained from $\pi_0$, as the resulting matrix is identical):

$$\dfrac{\dfrac{\dfrac{\vdots\ \pi_1}{\vdash \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & p & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & \infty & 0 \\ 0 & p & m \end{pmatrix}}\ \mathrm{L}^\infty \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\ \mathrm{E2}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & w & 0 \\ 0 & w & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & w & 0 \\ 0 & \infty & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{L}^\infty$$

Of course, neither of those two derivations would yield polynomial bound – since they contain $\infty$ coefficients – but it becomes possible to determine that the last one is "better" – since $\begin{pmatrix} p \\ \infty \\ p \end{pmatrix} > \begin{pmatrix} w \\ \infty \\ 0 \end{pmatrix}$ – and to observe how their "failure" would propagate in larger programs, possibly establishing that one fares better than the other in terms of non-polynomial growths. This could imply, for instance, that particular programs without polynomial bounds could still be considered "reasonnable" if they are exponential only in some variables that are known to have smaller values in input.

## 3.3 Merging the improvements: illustrations and proofs

We prove that our system captures the original system in the sense that set aside $\infty$ coefficients, both systems agree (Theorem 11), but also that exactly one matrix is produced per program (Theorem 10) – i.e., that we can analyze as many programs as originally, and still be correct regarding the bounds. Before doing so, we would like to give more specifics on our system, by combining the semi-rings and intuitions from the previous two subsections. We have discussed our "axiomatic" ($E^A$, $E^M$, $E^S$) and "loop" rules ($L^\infty$ and $W^\infty$), but remain to discuss the rules for assignment (A), `if` (I) and composition (C) – which is where both improvements meet. Mathematically speaking, adopting the semi-ring defined over matrices with coefficients in $\{0,1,2\}^p \to \text{MWP} \cup \{\infty\}$ is straightforward, and we simply write $\oplus$ and $\otimes$ the operations resulting from merging the two transformations. We discuss in Sect. 4.3 how, however, those operations are computationally costly and how we address this challenge.

▶ **Example 8.** Using our deterministic system presented in Fig. 2, consider the following:

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : V}\ \mathrm{E}^\mathrm{A}}{\vdash \texttt{X1 = X1 + X2} : 1 \overset{1}{\leftarrow} V}\ \mathrm{A} \qquad \dfrac{\dfrac{}{\vdash \texttt{X1 - X3} : V'}\ \mathrm{E}^\mathrm{A}}{\vdash \texttt{X1 = X1 - X3} : 1 \overset{1}{\leftarrow} V'}\ \mathrm{A}}{\vdash \texttt{if b then \{X1 = X1 + X2\} else \{X1 = X1 - X3\}} : (1 \overset{1}{\leftarrow} V) \oplus (1 \overset{1}{\leftarrow} V')}\ \mathrm{I}$$

with

$$\begin{aligned}
V &= 0 \mapsto \{{}^m_1, {}^p_2\} \oplus 1 \mapsto \{{}^p_1, {}^m_2\} \oplus 2 \mapsto \{{}^w_1, {}^w_2\} \\
V' &= 0 \mapsto \{{}^m_1, {}^p_3\} \oplus 1 \mapsto \{{}^p_1, {}^m_3\} \oplus 2 \mapsto \{{}^w_1, {}^w_3\} \\
1 \overset{1}{\leftarrow} V &\cong \begin{pmatrix} (0{\mapsto}m)\oplus(1 \mapsto p)\oplus(2{\mapsto}w) & 0 & 0 \\ (0{\mapsto}p)\oplus(1 \mapsto m)\oplus(2{\mapsto}w) & m & 0 \\ 0 & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & 0 \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \\
1 \overset{1}{\leftarrow} V' &\cong \begin{pmatrix} (0{\mapsto}m)\oplus(1 \mapsto p)\oplus(2{\mapsto}w) & 0 & 0 \\ 0 & m & 0 \\ (0{\mapsto}p)\oplus(1 \mapsto m)\oplus(2{\mapsto}w) & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0,1)\oplus p\delta(1,1)\oplus w\delta(2,1) & 0 & 0 \\ 0 & m & 0 \\ p\delta(0,1)\oplus m\delta(1,1)\oplus w\delta(2,1) & 0 & m \end{pmatrix}
\end{aligned}$$

Some care is needed to perform the addition for the I rule: the choices in the left and right branches are independent, so we must use coefficients in $\{0,1,2\}^2 \to \textsc{mwp}$ for the $2^3$ choices. While the mapping notation would require to use positions to describe which choice is being refereed to, the $\delta$ notation makes it immediate, as it encodes in the second value of $\delta$ that two choices are considered, numbering the choice in the left branch 0. Hence we can sum the coefficients and obtain the matrix that can be observed in our implementation by analyzing `example7.c`.

▶ **Example 9.** Our deterministic system now assigns to `loop X3 {X2 = X1 + X2}` from Example 3 the unique matrix

$$\begin{pmatrix} m & (0 \mapsto p) \oplus (1 \mapsto m) \oplus (2 \mapsto w) & 0 \\ 0 & (0 \mapsto m) \oplus (1 \mapsto \infty) \oplus (2 \mapsto \infty) & 0 \\ 0 & (0 \mapsto p) \oplus (1 \mapsto 0) \oplus (2 \mapsto 0) & m \end{pmatrix} = \begin{pmatrix} m & p\delta(0,0) \oplus m\delta(1,0) \oplus w\delta(2,0) & 0 \\ 0 & m\delta(0,0) \oplus \infty\delta(1,0) \oplus \infty\delta(2,0) & 0 \\ 0 & p\delta(0,0) \oplus 0\delta(1,0) \oplus 0\delta(2,0) & m \end{pmatrix}$$

where we observe that

1. only one choice, one assignment, 0, gives a matrix without $\infty$ coefficient, corresponding to the fact that, in the original system, only $\pi_0$ could be used to complete the proof,
2. the choice impacts the matrix locally, the coefficients being mostly the same, independently from the choice,
3. the influence of `X2` on itself is where possible non-polynomial growth rates lies, as the $\infty$ coefficient are in the second column, second row.

We are now in possession of all the material and intuitions needed to state the correspondence between our system and the original one of Jones and Kristiansen.

▶ **Theorem 10** (Determinancy and termination). *Given a program $P$, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \textsc{mwp}^\infty)$ such that $\vdash P : M$.*

**Proof.** The existence of the matrix is guaranteed by the completeness of the rules, as any program written in the syntax presented in Sect. 2.1 can be typed with the rules of Fig. 2. The uniqueness of the matrix is given by the fact that no two rules can be applied to the same command. Details are provided in Appendix B. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

**Proof.** The proof uses that $P$ cannot be assigned a matrix in the original calculus iff the deterministic calculus introduce a $\infty$ coefficient, and from the fact that both calculus coincide in all the other cases. Details are provided in Appendix B. ◀

▶ **Corollary 12** (Soundness). *If $\vdash P : M$ and there exists $\vec{a} \in A^p$ such that $\infty \notin M[\vec{a}]$, then every value computed by $P$ is bounded by a polynomial in the inputs.*

**Proof.** This is an immediate corollary of the original soundness theorem [23, Theorem 5.3] and of Theorem 11. ◀

This proves that the two analyses coincide, when excluding $\infty$, and that we can re-use the original proofs. However, our alternative definition should be understood as an important improvement, as it enables a better proof-search strategy while optimizing the memory usage, and hence enables the implementation (Sect. 5). It also lets the programmer gain more fine-grained feedback, and illustrates the flexibility of the analysis: the latter will also be demonstrated by the improvements we discuss in the next section.

# 4 Extending and improving the analysis: functions and efficiency

To improve this analysis, one could try to extract a tight bound, to certify it, or to port it to a compiler's intermediate representation. Adding constant values is arguably immediate [23, p. 3] but handling pointers, even if technically possible, would probably require significant work. This illustrates at the same time the flexibility of the analysis, and the distance separating ICC-inspired techniques from their usage on actual programs. We decided to narrow this gap along two axes: the first one consists of allowing function definitions and calls in our syntax. It is arguably a small improvement, but illustrates nicely the compositionality of the analysis, and includes recursively defined functions. The second extension intersects the theory and the implementation: it details how our semi-ring structure can be leveraged to maintain a tractable algorithm to compute costly operations on our matrices, and to separate the problem of deciding if a bound exists from computing its form.

## 4.1 Leveraging compositionality to analyze function calls

Thanks to its compositionality, this analysis can easily integrate functions and procedures, by re-using the matrix and choices of a program implementing the function called. We begin by adding to the syntax the possibility of defining multiple functions and calling them:

▶ **Definition 13** (Functions). *Letting `R` (resp. `f`) range over variables (resp. function names), we add* function calls[4] *to the commands (Def. 1) and allow* function declarations*:*

`C ::= Xi = f(X1, ..., Xn)`        `F ::= f(X1, ..., Xn){C; return R}`

*In a function declaration, `f(X1, ..., Xn)` is called the* header*, and the* body *is simply `C` (i.e., `return R` is not part of the body). A* program *is now a series of function declarations such that all the function calls refer to previously declared functions – we deal with recursive calls in Sect. 4.2 – and a* chunk *is a series of commands.*

Now, given a function declaration computing $f$, we can obtain the matrix $M_f$ by analyzing the body of $f$ as previously done. It is then possible to store the assignments $\vec{a}_0, \dots, \vec{a}_k$, for which no $\infty$ coefficients appear[5], and to project the resulting matrices to only keep the vector at `R` that provides quantitative information about all the possible dependencies of the output variable `R` w.r.t. input values, possibly merging choices leading to the same result. After this, we are left with a family $(M_f[\vec{a}_0])|_{\mathtt{R}}, \dots, (M_f[\vec{a}_k])|_{\mathtt{R}}$ of vectors – as the syntax here is restricted to functions with a single output value, even if accommodating multiple return values would be dealt with the same way – that we can re-use when calling the function.

The analysis of the command calling $f$ is then dealt with the F rule below:

$$\frac{}{\vdash \mathtt{Xi = F(X1,\dots, Xn)} : 1 \overset{\mathtt{i}}{\leftarrow} (((M_f[\vec{a}_0])|_{\mathtt{R}})\delta(0,c) \oplus \cdots \oplus ((M_f[\vec{a}_k])|_{\mathtt{R}})\delta(k,c))}\ \text{F}$$

This rule introduces a choice $c$ over $k$ possible matrices, and it is possible that $k \neq 3$, but this is not an issue, since our semi-ring construction can accommodate any set of choice $A$.

[4] Function calls that discard the output – procedures – could also be dealt with easily, but are vacuous in our effect-free, in particular pointer-free, language

[5] Allowing $\infty$ coefficients would not change the method described nor its results, but it does not seem relevant to allow calling functions that are not polynomially bounded.

▶ **Example 14.** Consider the following two programs `Q` and `P`:

```
Q =  int f(X1, X2){
       while b do {X2=X1+X1};
       return X2;
     }
```

```
P =  int foo(X1, X2){
       X2=X1+X1;
       X1=f(X2, X2);
     }
```

We first have $\vdash$ `X2 = X1 + X1` : $V$ for $V = \begin{pmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & 0 \end{pmatrix}$, and since $V^* = \begin{pmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{pmatrix}$, applying $\mathrm{W}^\infty$ gives $\vdash$ `Q` : $\begin{pmatrix} m & \infty\delta(0,0)\oplus \infty\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{pmatrix}$. Noting that only one choice gives an $\infty$-free matrix, we can now carry on the analysis of `P`:

$$\dfrac{\dfrac{\vdots}{\vdash \texttt{X2 = X1 + X1} : V} \quad \dfrac{}{\vdash \texttt{X1 = f(X2, X2)} : 1 \xleftarrow{1} ((\begin{smallmatrix} w \\ m \end{smallmatrix})\,\delta(0,c))}\ \mathrm{F}}{\vdash \texttt{P} : V \otimes 1 \xleftarrow{1} ((\begin{smallmatrix} w \\ m \end{smallmatrix})\,\delta(0,c))}\ \mathrm{C}$$

In this particular case, the $c$ choice can be discarded, since only one option is available.

Now, to prove that the F rule faithfully extends the analysis (Theorem 17), i.e., preserves Corollary 12, we prove that the analysis of the program "inlining" the function call – as defined below – is, up to some bureaucratic variable manipulation and ignoring some $\infty$ coefficients, the same as the analysis resulting from using our rule. Intuitively, this mechanism provides the expected result because the choices in the function *do not* affect the program calling it, and because their sets of variables are disjoint – except for the return variable.

▶ **Definition 15** (In-lining function calls). *Let $P$ be a chunk containing a call to the function $f$, and $F$ be the function declaration computing the function $f$. The* context *$P[\cdot]$, a chunk containing a slot $[\cdot]$, is obtained by replacing in $P$ the function call `Xi=f(X1, ..., Xn)`, with `X'1=X1; ...; X'n=Xn;` $[\cdot]$ `Xi=R`, for `R`, `X'1, ..., X'n` fresh variables added to the header containing the chunk.*

*The chunk $\tilde{F}$ is obtained from the body of $F$ by renaming the input variables to `X'1, ..., X'n`, and the variable returned by $F$ to `R`. The code $P[F]$ is finally obtained by computing the chunk $\tilde{F}$, and inserting it in place of the symbol $[\cdot]$ in $P[\cdot]$.*

That $P$ and $P[F]$ have, at the end of their executions, the same values stored in the variables of $P$ is straightforward in our imperative programming language.

▶ **Example 16.** The in-lining of `Q` in `P` from Example 14 would give the following chunk $\tilde{\texttt{Q}}$ and context `P`$[\cdot]$, `P[Q]` being obtained by replacing in the latter $[\cdot]$ with the former:

$\tilde{Q} =$ `while b do {R=X'1+X'1};`

$P[\cdot] =$

```
int foo(X1, X2, X'1, R){
    X2=X1+X1;
    X'1=X2;
    [·]
    X1=R;
}
```

The analysis of `P` (excluding the function call) and `Q` is implemented at `example15a.c`, and of `P[Q]` at `example15b.c`: this latter diverges with Example 14 only up to projection and $\infty$-coefficients that are removed by F but not when in-lining the function call.

Now, we need to prove that the matrices $M(P)$ – obtained by analyzing $P$ and using the F rule for `Xi=f(X1, ..., Xn);` – and $M(P[F])$ – obtained by analyzing the inlined $P[F]$ – are the same. However, to avoid conflict with the variables and to project the matrices on the relevant values, some bureaucracy is needed: we write $\Pi_P(M(P[F]))$ (resp. $(1 - \Pi_P)(M(P[F]))$) the projection of $M(P[F])$ onto the variables in (resp. *not* in) $P$. Some non-deterministic choices may appear within the (modified) chunk $\tilde{F}$ inside $P[F]$, i.e.,

- the coefficients of $M(P)$ are elements of the semi-ring $\prod_{i=1}^{p+1} A_i \to \mathbb{M}(\textsc{mwp})$, with one particular choice corresponding to the F rule – we write the corresponding index $i_0$;
- the coefficients of $M(P[F])$ are elements of the semi-ring $\prod_{i=1}^{p+k} B_i \to \mathbb{M}(\textsc{mwp})$, where $k$ choices are made within the chunk $\tilde{F}$ – we write the corresponding indexes $j_1, j_2, \dots, j_k$ (note these are in fact consecutive indexes).

We note $\pi : \{1, \dots, p+k\} \to \{1, \dots, p+1\}$ the projection of the choices in $P[F]$ onto the corresponding choices in $P$, i.e., $\pi(j) = \begin{cases} j & \text{if } j < j_1 \\ i_0 & \text{if } j_1 \leqslant j < j_k \\ j-k+1 & \text{if } j_k < j \end{cases}$. We note that each matrix used as axiom in the function call corresponds to a specific assignment on indexes $j_1, \dots, j_k$. We write $\Psi : A_{i_0} \to \prod_{i=j_1}^{j_k} B_i$ the corresponding injection, extended to $\bar{\Psi} : \prod_{i=1}^{p+1} A_i \to \prod_{i=0}^{p+k} B_i$ straightforwardly.

▶ **Theorem 17.** *For all $\vec{a}$ in $\prod_{i=1}^{p+1} A_i$, $(M(P))[\vec{a}] = (1 - \Pi_P)(M(P[F]))[\bar{\Psi}(\vec{a})]$, and for all $\beta$ in $\prod_{i=0}^{p+k} B_i$ not in the image of $\bar{\Psi}$, $(1 - \Pi_P)(M(P[F])[\beta])$ contains $\infty$.*

**Proof.** It is sufficient to prove it for the simplest chunk $P$ containing only one command `Xi = f(X1, ..., Xn)`. This comes from the compositional nature of the analysis, as a sequence of commands is assigned the product of the matrices of each individual command. Then, checking the theorem in this case is a straightforward, though tedious (due to keeping track of all indices), computation. ◀

## 4.2 Integrating recursive calls, the easy way

The question of dealing with self-referential, or recursive, calls, naturally arises when extending to function calls. It turns out that our approach makes such cases easy to handle.

A program implementing a function `rec` calling itself cannot use the F rule presented above as is, since the result of the analysis of `rec` is precisely what we are trying to establish. However, if `rec` takes two input variables `X1` and `X2` and its return value is assigned to a third variable `X3`, then we already know that the vector at 3 will need to be replaced by the vector capturing the dependency between `X1`, `X2`, and the return variable of `rec` (which we will take to be `X3` in our example). The solution consists in replacing the actual values in this vector by variables $\alpha$, $\beta$ ranging over values in $\textsc{mwp}^\infty$, terminating the analysis with those variables, and then to resolve the equation – which is easy given the small size of the $\textsc{mwp}^\infty$ semiring.

As an example[6], consider the following program and compute the corresponding matrix:

```
int rec(X1, X2){
  X1 = X1 + X2;
  X3 = rec(X1, X2);
  return X3;
}
```

$$\begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & 0 \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \otimes 1 \xleftarrow{3} \begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & \alpha m\delta(0,0)\oplus\alpha p\delta(1,0)\oplus\alpha w\delta(2,0) \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & \alpha p\delta(0,0)\oplus\alpha m\delta(1,0)\oplus\alpha w\delta(2,0)\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$$

Using the assignments 0, 1 and 2 gives $\begin{pmatrix} m & 0 & \alpha m \\ p & m & \alpha p\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, $\begin{pmatrix} p & 0 & \alpha p \\ m & m & \alpha m\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$ and $\begin{pmatrix} w & 0 & \alpha w \\ w & m & \alpha w\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, and since the third vector should be equal to $\begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$, this gives three systems of equations:

[6] Where we use variables that are not parameters, following footnote 1, and where our recursive call does not terminate: we are focusing on growth rates and not on termination, and keep the example compact.

$$\begin{cases} \alpha m &= \alpha \\ \alpha p \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha p &= \alpha \\ \alpha m \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha w &= \alpha \\ \alpha w \oplus \beta &= \beta \end{cases}$$

The smaller solution to the first (resp. second, third) equational system is $\{\alpha = m; \beta = p\}$ (resp. $\{\alpha = p; \beta = p\}$ , $\{\alpha = w; \beta = w\}$), and as a consequence, we find two meaningful solutions (all others being larger than those): $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

## 4.3 Taking advantage of polynomial structure to compute efficiently

Ensuring that the analysis is tractable is an important part of our contribution. For a program accepting $n$ different derivations and having $k$ different derivations that cannot be completed, the original flow calculus must run at most $k + 1$ times to find *one* derivation, while our analysis outputs the $k + n$ different derivations in one run, and then sorts them – as discussed next – by listing all the evaluations and looking for $\infty$ values. In this task, the C rule, that lets building programs from commands, is obviously crucial and consists simply in multiplying two matrices: however, since we are internalizing the choices, those matrices contain a mixture of functions from choices to coefficients in $\text{MWP}^\infty$ and of coefficients in MWP. Multiplying such matrices is more costly, but also essential: an 8-line program such as `explosion.c` requires to multiply elements of its matrix 34,992 times[7]. This forces to represent and manipulate the elements of $\prod_{i=1}^{p} A_i \to \mathbb{M}(\text{MWP})$ – setting aside $\infty$ coefficients for a moment – cleverly: simple comparison showed that the improved algorithm presented below made the analysis roughly *five times* faster (Sect. C.3).

As discussed in Notation 5, elements of this semi-ring are represented as *polynomials* w.r.t. the generating set given by the functions $\delta(i, j) : \prod_{i=1}^{p} A_i \to \text{MWP}$ defined by $\delta(i, j)(a_1, \dots, a_p) = m$ if $a_j = i$ and $\delta(i, j)(a_1, \dots, a_p) = 0$ otherwise, i.e., an element of $\prod_{i=1}^{p} A_i \to \text{MWP}$ is represented as a polynomial $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ with $\alpha_i \in \text{MWP}$.

This basis has an important property: the *monomials* $\alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ in a polynomial can be ordered so that the product with another monomial is ordered, i.e., if $\alpha \leqslant \beta$ and both $\alpha \times \gamma$ and $\beta \times \gamma$ are non-zero, then $\alpha \times \gamma \leqslant \beta \times \gamma$. This order is leveraged to obtain efficient algorithms, similar to what is done using Gröbner bases for computation of standard polynomials [35]. For instance, the algorithm for multiplication of polynomials uses this property to compute the product of an ordered polynomial $P$ with $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$:

1. compute the products $P_i = P \times \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ for all $i$;
2. compare and order a list $L$ of all the first elements of those polynomials;
3. append the smallest element to the result and remove it from the corresponding $P_i$;
4. insert the (new) first element of $P_i$ to the list $L$ if it exists;
5. if $L$ is non-empty, go back to step 3.

When adding or multiplying polynomials, which consist of monomials, we check if a monomial is contained or included by another, and exclude all redundant cases (cf. `contains` or `includes`). This is also done when inserting monomials. Thus we keep polynomials free of implementation choices that we would otherwise have to handle during evaluation.

[7] The need to optimize functions is made even more obvious when we discuss benchmarking in Sect. 5.1.

## 4.4 Deciding the existence of a bound faster thanks to delta graphs

Adopting the $\prod_{i=1}^{p} A_i \to \textsc{mwp}^\infty$ semi-ring permits to complete all derivations simultaneously, but remains to determine if there exists an assignment $\vec{a} \in \prod_{i=1}^{p} A_i$ s.t. the resulting matrix is $\infty$-free, to decide whenever a program accepts a polynomial bound: this is the *evaluation* step. Despite the optimizations detailed above that simplifies the task, this phase remains particularly costly, since the number of assignment grows exponentially w.r.t. the number of choice, which is linear in the number of variables. While this step is necessary (in one form or another) if one wishes to produce the actual mwp matrices certifying polynomial bounds, we implemented a specific data structure to keep track of assignments resulting in $\infty$ coefficients on the fly, thus allowing the analysis to provide a qualitative answer quickly. This section details how those *delta graphs* allow to immediately determines whenever a polynomial bound exists without having to compute the corresponding matrix, something that was not possible in the original, non-deterministic, calculus.

A delta graph is a graph whose vertices are monomials. The graph is populated during the analysis by adding those monomials that appear with an infinite coefficient – i.e., possible choices leading to $\infty$ in the resulting matrix. This graph is structured in layers: each layer corresponds to the size of the monomials (the number of deltas) it contains. The intuition is that a monomial – or rather a list of deltas $\delta(_,_)$ – defines a subset of the space $\prod_{i=1}^{p} A_i$; the less deltas in the monomial, the greater the subspace represented[8]. As we populate the delta graph, we create edges within a given layer to keep track of differences between monomials: we add an edge labeled $i$ between two monomials if and only if they differ only on one delta $\delta(_, i)$ (i.e., one is obtained from the other by replacing the first index of $\delta(_, i)$). This is used to implement a `fusion` method on delta graphs, which simplifies the structure: as soon as a monomial $m$ in layer $n$ has $\mathrm{Card}(A_i) - 1$ outgoing edges labelled $i$, we can remove all these monomials and insert a shorter monomial in layer $n - 1$, obtained from $m$ by simply removing $\delta(_, i)$. This implements the fact that $\sum_{k=0}^{\mathrm{Card}(A_i)-1} m\delta(k, j) = m$.

Now, remember the delta graph represents the subspace of assignments for which an $\infty$ appears. If at some point the delta graph is completely simplified (i.e., "fusions" to the graph with a unique monomial consisting in an empty list of $\delta(_,_)$), it means the whole space of assignments is represented and no mwp-bounds can be found. On the contrary, if the analysis ends with a delta graph different from the completely simplified one, at least one assignment exists for which no infinite coefficients appear, and therefore at least one mwp-bound exists. This allows one to answer the question "Is there at least one mwp-bound?" *without actually computing said bounds*. Based on the information collected in the delta graph and the matrix with polynomial coefficients, one can however recover all possible matrix assignments by going through all possible valuations.

This last part is implemented with a specific iterator that leverages the information collected in the delta graph to skip large sets of valuations in a single step. For instance, suppose the monomial $\delta(1, 1)$ lies in the delta graph – i.e., that an infinite coefficient will be reached if the second index is equal to 1. When asked the valuation after $(0, 0, 2, 2)$ (and supposing that $\mathrm{Card}(A_i) = 3$ for all $i$), our `delta_iterator` will jump directly to $(0, 2, 0, 0)$, skipping all intermediate valuation of the form $(0, 1, a, b)$ in a single step. Similarly, it will jump from $(1, 0, 2, 2)$ to $(1, 2, 0, 0)$, again skipping several valuations at a time, providing a

[8] Our intuitions here come from the standard topological structure of spaces of infinite sequences, where such a monomial represents a "cylinder set", i.e., an element of the standard basis for open sets.

faster analysis. Note that the implementation required care, to correctly jump when given additional informations from the delta graph, e.g., to produce $(2, 0, 1, 0)$ as the successor of $(0, 0, 2, 2)$ if $\delta(0, 0)$, $\delta(1, 1)$ and $\delta(0, 2)$ all belong to the delta graph.

# 5 Implementing, testing and comparing the analysis

Demonstrating the implementability of the improved and extended mwp-bounds analysis requires an implementation. Our open-source solution, packaged through Python Package Index (PyPI) as `pymwp`, is a standalone command line tool, written in `Python`, that automatically performs growth-rate analysis on programs written in a subset of the `C` programming language. For programs that pass the analysis, it produces a matrix corresponding to the input program and a list of valid derivation choices; and for programs that do not have polynomial bounds, it reports infinity. Our motivation for choosing `C` as the language of analysis resulted from its central role and similarity with the original `while` language. `Python` was an ideal choice for the implementation because of its plasticity, collection of libraries, and because it allowed partial reuse of a previous flow analysis tool [3, 31, 32]. The source code is available on Github, along with an online demo, and detailed documentation [33] describing its current supported features and functionality. We now discuss how we tested and assesed it, and how it compares (or, rather *does not* compare) to other similar approaches.

## 5.1 Experimental evaluation

We allocated extensive focus and effort on testing and profiling our implementation, to ensure the correctness and efficiency of the analysis, and with the terminal objective of obtaining a usable tool. The test suite includes 42 `C` programs, carefully designed to exercise different aspects of the analysis, ranging from basic derivations, to ones producing worst-case behavior (by yielding e.g., dense matrices or exponential number of derivations), and classical examples such as computing the greatest common divisor or exponentiation.

We refer to our benchmarks (presented in Appendix C) for measured analysis results for each program. The most salient aspect is that our analysis is extremely fast (the time is measured in *milli*seconds) despite important numbers of function calls (in the 10k range, excluding builtin Python language calls, for 10-lines programs). Even examples tailored to stress our implementation cannot make the analysis go over *4 seconds*. We cannot compare our implementation with implementations of the original analysis, since it has never been implemented, and (according to our attempts) cannot be implemented in any realistic manner.

## 5.2 Related tools and incompatible metrics

This work was inspired by the series of works of the flow analysis from the "Copenhagen school" [11, 24]. The overall flow analysis approach is related in spirit to abstract interpretation [13, 14]; that bounds *transitions* between states (e.g., commands) instead of states [24]. This approach shaped the implementation of tools detecting loop quasi-invariants [31, 32].

Other communities share a similar goal of inferring resource-usage. Complexity analyzers such as SPEED [19] for `C++`, COSTA [1] for `Java` bytecode, ComplexityParser [21] for `Java`, Resource Aware ML for `OCaml` [29] or Cerco [2] and Verasco [25] for `C` generate (certified) cost or runtime analysis on (subsets of) imperative programming languages. Embracing such a large diversity is difficult, but our technique is different from existing implementations and tools: most of them focus on worst-case resource-usage complexity or termination, while we

are interested in upper-bounds on the final values of program variables, i.e., we focus on *growth* instead of actual values. This makes the comparison with our approach difficult, but highlights at the same time its uniqueness in today's landscape of static analyzers.

Further, our approach provides other desirable properties:

1. it is compositional, which allows one to "hot-plug" bounds of previously analyzed functions without additional work,
2. it is modular, as the internal machinery can be altered – as in this paper – without having to re-develop the theory,
3. it is language-independent, as it reasons abstractly on imperative languages, but can be applied to real programs, as our implementation illustrates, and should extend to more complex languages,
4. it is lightweight and programmer-friendly, as it is fast, does not require annotations or to record value ranges,
5. it studies growth independently from e.g., iteration bounds, thus sidestepping difficult cases that worst-case analysis has to tackle, and
6. it may enable tight bounds on programs, as it has been done recently [10] for a similar analysis [11].

In particular compositionality is a highly desirable property – because otherwise the analysis needs to be re-run on programs or API whenever embedded into different pieces of software – yet difficult to achieve by most other approaches, as discussed and partially remedied recently [12]. While we suppose one approach could be used to derive the result obtained by the other, we do believe the originality of our pioneering ICC-based approach may inspire new and original directions in static program analysis.

## 6 Conclusion: limitations, strengths and future work

This work attempts to illustrate the usefulness and applicability of ICC results, but also the need to refine and adapt them. We showed that the mwp-flow analysis as originally described cannot scale to programs in a real programming language: while the considered analysis is definitely powerful and elegant, its mathematical nature let some costly operations go unchecked. However we have shown that, extended and coupled to optimizations techniques, its result enable the development of a novel and original static analysis technique on imperative programs, focused on *growth* rather than on termination or worst-case bounds.

This work is a proof of concept and it has limitations, both theoretical and practical: the theory is missing memory uses, pointers, and arrays and the supported feature set of the implementation could be extended. But instead of focusing on what this analysis *cannot* perform, we would like to stress that all the tools are in place to perform similar analysis on intermediate representations of code in compilers, which will naturally simplify the task of fitting richer program syntax to our analysis, and brings this technique yet another step closer to practical use cases.

One of our next steps include certifying the analysis using the Coq proof assistant [34], and implementing the analysis in certified tools such as the Compcert compiler [28] (or, more precisely, its static single assignment version [7]) or certified-llvm [36]. The plasticity of both compilers and of the implemented analysis should facilitate porting our results and approaches to support further programming languages in addition to `C`. As complexity analysis is notably difficult in Coq [20], we believe a push in this direction would be welcome, and that ICC provides all the needed tools for it.

Another direction is to explore the possibility for our analysis to focus on the *final* values of variables instead of tracking them throughout the whole program. Indeed, recall from Sect. 3.2 that our semi-ring is such that $\infty \times^{\infty} 0 = \infty$, but another valid choice would have been to pick $\infty \times^{\infty} 0 = 0$ [4, A.4]. In this case, it seems that if some non-polynomial growth is caused by a variable that is then "thrown away" (overridden), then a program could still pass the analysis: whether this lead gives relevant results is yet to determine, but it would be another nice illustration of the plasticity of this analysis.

Last but not least, working on finer comparison with other static analyzers [18] could be useful. We have stressed in Sect. 5.2 how such comparison was uneasy, as the finality of our tool is not directly comparable with any other analyzer we know of. However, some tools such as AProVE [17] or CoFloCo [16] provide polynomial upper bounds for `C` programs, and we could assess e.g., on the Termination Problems Data Base whether `pymwp` can analyze as many problems and how often all three analyzers agree. The motivation for the original mwp-analysis was to develop resource analysis for distinguishing feasible problems and to work at the boundary of undecidability, but this could actually be one of `pymwp`'s strength as a *pre-processor* to other static analyzers, to save them from running costly analysis on programs known to be unfeasible. This fast analysis and the compositionality of our tool could also, on longer term, be useful to construct IDE plug-ins that provide low-latency feedback to the programmer.

## References


**1** Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *LNCS*, pages 113–132. Springer, 2007. `doi:10.1007/978-3-540-92188-2_5`.

**2** Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, volume 8552 of *LNCS*, pages 1–18. Springer, 2013. `doi:10.1007/978-3-319-12466-7_1`.

**3** Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Lqicm on c toy parser. URL: `https://github.com/statycc/LQICM_On_C_Toy_Parser`.

**4** Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. Preliminary technical report, March 2022. URL: `https://hal.archives-ouvertes.fr/hal-03596285`.

**5** Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP):43:1–43:29, 2017. `doi:10.1145/3110287`.

**6** Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. `doi:10.1109/LICS.2004.1319621`.

**7** Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, 2014. `doi:10.1145/2579080`.

**8** Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. `doi:10.1145/129712.129740`.

**9** Amir M. Ben-Amram. On decidable growth-rate properties of imperative programs. In Patrick Baillot, editor, *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010*, volume 23 of *EPTCS*, pages 1–14, 2010. doi:10.4204/EPTCS.23.1.

**10** Amir M. Ben-Amram and Geoff W. Hamilton. Tight polynomial worst-case bounds for loop programs. *Log. Meth. Comput. Sci.*, 16(2), 2020. doi:10.23638/LMCS-16(2:4)2020.

**11** Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann and Costas Dimitracopoulos andBenedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability inEurope, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008. doi:10.1007/978-3-540-69407-6_7.

**12** Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 467–478. ACM, 2015. doi:10.1145/2737924.2737955.

**13** Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

**14** Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada, August 1-5, 1977*, pages 237–278. North-Holland, 1977.

**15** Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. doi:10.1007/978-3-642-31485-8_3.

**16** Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *LNCS*, pages 254–273, 2016. doi:10.1007/978-3-319-48989-6_16.

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

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

**19** Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480898.

**20** Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02437532.

**21** Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In Antonio Cerone and

Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *LNCS*, pages 357–365. Springer, 2021. doi:10.1007/978-3-030-85315-0_20.

22 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012. doi:10.1007/978-3-642-31424-7_64.

23 Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. doi:10.1145/1555746.1555752.

24 Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In Samson Abramsky, Dov M. Gabbay, and Thomas Stephen Edward Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 527–636. Oxford University Press, 1995.

25 Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium onPrinciples of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

26 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1):163–180, 2004. doi:10.1016/j.tcs.2003.10.018.

27 Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 325–333. ACM Press, 1993. doi:10.1145/158511.158659.

28 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

29 Benjamin Lichtman and Jan Hoffmann. Arrays and references in resource aware ML. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 26:1–26:20. Schloss Dagstuhl, 2017. doi:10.4230/LIPIcs.FSCD.2017.26.

30 Jean-Yves Moyen. *Implicit Complexity in Theory and Practice.* Habilitation thesis, University of Copenhagen, 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

31 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In Deepak D'Souza and K. Narayan Kumar, editors, *ATVA*, volume 10482 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-68167-2_7.

32 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk motion by peeling with statement composition. In Guillaume Bonfante and Georg Moser, editors, *Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017*, volume 248 of *EPTCS*, pages 47–59, 2017. doi:10.4204/EPTCS.248.9.

33 pymwp's documentation, 2021. URL: https://statycc.github.io/pymwp/.

34 Coq Team. Coq documentation, 2022. URL: https://coq.github.io/doc/.

35 Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, 30(6):509–539, December 2019. doi:10.1007/s00200-019-00389-9.

36 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 175–186. ACM, 2013. doi:10.1145/2491956.2462164.

# A Technical appendix on semi-rings (abridged)

This is an abridged version of the technical development on semi-ring that is exposed in full details in our technical report [4, A.1 and A.2].

▶ **Lemma 18** (mwp semi-ring). *The tuple* $(\{0, m, w, p\}, 0, m, +, \times)$, *with*

- $0 < m < w < p$,
- $\alpha + \beta = \begin{cases} \alpha & \text{if } \alpha \geqslant \beta \\ \beta & \text{otherwise} \end{cases}$
- $\alpha \times \beta = \begin{cases} \alpha + \beta & \text{if } \alpha \neq 0 \text{ and } \beta \neq 0 \\ 0 & \text{otherwise} \end{cases}$

*is a* strong *semi-ring.*

▶ **Lemma 19** (Matrix semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$, *the tuple* $\mathbb{M} = (M, 0, 1, \oplus, \otimes)$, *with*

- $M$ *the set of all* $n \times n$ *matrices over* $S$, *for all* $n \in \mathbb{N}$,
- $0$ *defined by* $M = 0$ *iff* $M_{ij} = 0$ *for all* $i$ *and* $j$,
- $1$ *defined by* $M = 1$ *iff* $M_{ij} = 1$ *for* $i = j$, $M_{ij} = 0$ *otherwise,*
- $\oplus$ *defined by* $C = A \oplus B$ *iff* $C_{ij} = A_{ij} + B_{ij}$,
- $\otimes$ *defined by* $C = A \otimes B$ *iff* $C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$,

*is a strong semi-ring.*

For simplicity, we will write $\mathbb{M}$ as $\mathbb{M}(\mathbb{S}) = (M(S), 0, 1, \oplus, \otimes)$.

▶ **Lemma 20** (Choices semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and a set* $A$, *the tuple* $\mathbb{F} = (F, 0, 1, \boxplus, \boxtimes)$, *with*

- $F$ *the set of functions from* $A$ *to* $S$,
- $0$ *the constant function* $0(a) = 0$ *for all* $a \in A$,
- $1$ *the constant function* $1(a) = 1$ *for all* $a \in A$,
- $\boxplus$ *defined componentwise:* $(f \boxplus g)(a) = (f(a)) + (g(a))$, *for all* $f$, $g$ *in* $F$ *and* $a \in A$,
- $\boxtimes$ *defined componentwise:* $(f \boxtimes g)(a) = (f(a)) \times (g(a))$, *for all* $f$, $g$ *in* $F$ *and* $a \in A$,

*is a strong semi-ring.*

For simplicity, we will write $\mathbb{F}$ as $A \to \mathbb{S} = (A \to S, 0, 1, +, \times)$.

▶ **Lemma 21.** *For all set* $A$ *and strong semi-ring* $\mathbb{S}$, $\mathbb{M}(A \to \mathbb{S}) \cong A \to \mathbb{M}(\mathbb{S})$.

▶ **Lemma 22.** *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and an element* $\bot \notin S$, $\mathbb{S}^\bot = (S \cup \{\bot\}, 0, 1, +^\bot, \times^\bot)$ *with, for all* $a$, $b \in S \cup \{\bot\}$,

$$a +^\bot b = \begin{cases} a + b & \text{if } a, b \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$a \times^\bot b = \begin{cases} a \times b & \text{if } a, b \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

*is a semi-ring.*

**Proof.** The proof is immediate, but note that $\mathbb{S}^\bot$ is not strong, as $\bot \times 0 = \bot$. ◀

## B Omitted Proofs

▶ **Theorem 10** (Determinancy and termination). *Given a program $P$, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \text{MWP}^\infty)$ such that $\vdash P : M$.*

**Proof.** The proof proceeds by induction on the length of the program $P$, expressed in number of commands. We let $p$ be the number of variables in $P$, but observe that any program $P$ can be treated as manipulating $p' > p$ different variables, by simply adding $p' - p$ additional rows and columns to the matrix, and leaving them unchanged by the derivation of $P$. While a complete proof would need to constantly account for the number of actual and potential variables used by $P$, we will simply assume that the reader understands that accounting for this technicality obfuscate more than it clarifies the proof, and we will freely resize the matrices to account for additional variables when needed.

**If $P$ is of length 1** Then we know $P$ is of the form `X = e`, and only the rule A can be applied. But then we need to prove that all expression `e` can be typed with exactly one vector. An expression `e` is either a variable `X`, or a composed expression `X * Y`, `X - Y`, or `X + Y`. But then, respectively, only $\text{E}^\text{S}$, $\text{E}^\text{M}$ or $\text{E}^\text{A}$ (for addition and substraction) can be applied, and this case is proven.

**If $P$ is of length $n > 1$** Then we proceed by case on the structure of the command:

- If $P$ is of the form `if b then P1 else P2`, then by induction we know for $i \in \{1, 2\}$ there exists $p_i$ and $M_i$ of size $p_i \times p_i$ such that $\vdash$ `Pi` $: M_i$. If $p_1 \neq p_2$, then letting `Mj` being the smaller matrix, it is easy to rewrite `Pj`'s derivation to account for $|p_1 - p_2|$ additional variables, and as $\oplus$ is uniquely defined, we know that $M_1 \oplus M_2$ results in a unique matrix of size $\max(p_1, p_2)$.
- If $P$ is of the form `while b do P'`, this is immediate by induction hypothesis on `P'`, considering that only $\text{W}^\infty$ can be applied, and that this rule produces a unique matrix.
- If $P$ is of the form `loop X {P'}`, this case is similar to the previous one, using $\text{L}^\infty$ instead of $\text{W}^\infty$.
- If $P$ is of the form `P1;P2`, this case is similar to the `if` case, with the possible need to resize one of the matrix obtained by induction, and using that $\otimes$ is uniquely defined. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

**Proof.** The proof proceeds by induction on the length of the program $P$, expressed in number of commands.

**If $P$ is of length 1** Then we know $P$ is of the form `X = e`, and only the rule A can be applied, in both systems. Hence, we need to prove that all expression `e` can be typed the same way in both systems. A careful comparison of Figures 1 and 2 shows that if `e` is of the form `Xi`, then there is a small mismatch. In the original system, we can use either E2, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{w}_{\texttt{i}}\}$, or E1, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\texttt{i}}\}$, while the only derivation in the deterministic system is using $\text{E}^\text{S}$ to get $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\texttt{i}}\}$. As $m < w$, we argue that the deterministic system cannot obtain a derivation that is not useful anyway, and hence that it can be ignored.

As for the other cases, if `e` is a composed expression `X * Y`, `X - Y`, or `X + Y`, it is easy to observe that $\text{E}^\text{A}$ and $\text{E}^\text{M}$ encapsulates all the possible combinations of E2 and of E1 followed by E3 or E4 that can be used.

**If $P$ is of length $n > 1$** Then the result holds by induction, once we observed that $\text{L}^\infty$ and $\text{W}^\infty$ are introducing $\infty$ coefficients *only if* L and W cannot be applied. ◀

# C Benchmarks

## C.1 Descriptions of program groups

- *Basics* – C programs performing operations corresponding to simple derivation trees.
- *Implementation paper* – example programs presented in this paper.
- *Original paper* – examples taken from or inspired by the original analysis [23].
- *Infinite* – programs whose matrices always contain infinite coefficients.
- *Polynomial* – programs whose matrices do not always contain infinite coefficients.
- *Other* – other C programs of interest.

## C.2 Results

The benchmarks are categorized and grouped to distinguish the type of system behavior they exercise. For each program we capture in Table 1

1. program variable count
2. the lines of code in the source program (LOC column)
3. clock time taken by the full analysis (excluding saving result to file, which is otherwise default behavior),
4. number of function calls excluding builtin Python language calls, and
5. the result of the analysis.

Collectively the LOC, time, and function calls columns provide insight into the behavior of the analysis as different aspects of the system are being stress-tested. From the results column we report expected results on each benchmarked program. In the benchmarks table a passing result is represented with ✓and $\infty$ otherwise. We do not report manually computed bounds as comparison, because the analysis is carried out on individual variables, thus calculating them on multivariate programs is tedious and futile. However, for simple programs such as while_2.c, it is straightforward through visual inspection to verify the obtained $2 \times 2$-matrix is indeed the correct result.

These benchmarks were obtained using Python's built-in cProfile utility, extended in `pymwp` implementation to enable batch profiling. The clock times are slight overestimates because the utility adds minor runtime overhead. The number of function calls includes primitive calls, but exclude built-in Python language calls. Full detailed results are viewable in the source code repository: https://github.com/statycc/pymwp/releases/tag/profile-latest

## C.3 Comparison

It is not really meaningful or possible to compare those results with any other static analyzer, and impossible to compare it with any other implementation of this type of flow analysis. While we could, in theory, analyze our examples with other static analyzers, their results would be incomparable, as they would produce guarantees on termination or worst case resource usage, which are both orthogonal to our polynomial bounds on value growth. To our knowledge, the only static analyzer using similar metrics [5] was developed only for functional languages, thus preventing comparison. As for implementations of the original analysis, our first attempts showed that a naive implementation would likely fail to handle the memory or time explosions. We did, however, compare the gains resulting from the optimizations described in Sect. 4.3. In a nutshell, our improved algorithm for adding and multiplying polynomials resulted in the analysis being roughly *five times faster* for two programs that we estimate to be representative.

**Table 1** Benchmark results produced by `pymwp` on `C` programs.

| Program name | Variables | LOC | Time (ms) | Function calls | Bound |
|---|---|---|---|---|---|
| *Basics* | | | | | |
| assign_expression | 2 | 8 | 133 | 81614 | ✓ |
| assign_variable | 2 | 9 | 115 | 81238 | ✓ |
| if | 2 | 9 | 118 | 82046 | ✓ |
| if_else | 2 | 7 | 118 | 82928 | ✓ |
| inline_variable | 2 | 9 | 118 | 81979 | ✓ |
| while_1 | 2 | 7 | 117 | 82934 | ✓ |
| while_2 | 2 | 7 | 117 | 83964 | ✓ |
| while_if | 3 | 9 | 122 | 91572 | ✓ |
| *Implementation paper* | | | | | |
| example7 | 3 | 10 | 122 | 86898 | ✓ |
| example15_a | 2+2 | 25 | 122 | 88763 | ✓ |
| example15_b | 4 | 16 | 137 | 122016 | ✓ |
| *Original paper* | | | | | |
| example3_1_a | 3 | 10 | 110 | 85286 | ✓ |
| example3_1_b | 3 | 10 | 120 | 87637 | ✓ |
| example3_1_c | 3 | 11 | 121 | 89173 | ✓ |
| example3_1_d | 2 | 12 | 116 | 80002 | $\infty$ |
| example3_2 | 3 | 12 | 118 | 83182 | $\infty$ |
| example3_4 | 5 | 18 | 134 | 108890 | $\infty$ |
| example5_1 | 2 | 10 | 116 | 81185 | ✓ |
| example7_10 | 3 | 10 | 119 | 86053 | ✓ |
| example7_11 | 4 | 11 | 139 | 119379 | ✓ |
| *Infinite* | | | | | |
| exponent_1 | 4 | 16 | 127 | 99893 | $\infty$ |
| exponent_2 | 4 | 13 | 123 | 92846 | $\infty$ |
| infinite_2 | 2 | 6 | 143 | 128275 | $\infty$ |
| infinite_3 | 3 | 9 | 120 | 89880 | $\infty$ |
| infinite_4 | 5 | 9 | 3274 | 5924420 | $\infty$ |
| infinite_5 | 5 | 11 | 369 | 529231 | $\infty$ |
| infinite_6 | 4 | 14 | 1624 | 2836726 | $\infty$ |
| infinite_7 | 5 | 15 | 631 | 964189 | $\infty$ |
| infinite_8 | 6 | 23 | 880 | 1444782 | $\infty$ |
| *Polynomial* | | | | | |
| notinfinite_2 | 2 | 4 | 119 | 86174 | ✓ |
| notinfinite_3 | 4 | 9 | 131 | 104826 | ✓ |
| notinfinite_4 | 5 | 11 | 169 | 168242 | ✓ |
| notinfinite_5 | 4 | 11 | 174 | 176179 | ✓ |
| notinfinite_6 | 4 | 16 | 195 | 215765 | ✓ |
| notinfinite_7 | 5 | 15 | 1161 | 1961806 | ✓ |
| notinfinite_8 | 6 | 22 | 1893 | 3172293 | ✓ |
| *Other* | | | | | |
| dense | 3 | 16 | 157 | 151428 | ✓ |
| dense_loop | 3 | 17 | 269 | 353068 | ✓ |
| explosion | 18 | 23 | 1296 | 2327071 | ✓ |
| gcd | 2 | 12 | 114 | 84914 | $\infty$ |
| simplified_dense | 2 | 9 | 118 | 85098 | ✓ |

# Polynomial Termination Over ℕ Is Undecidable

**Fabian Mitterwallner**
Department of Computer Science, Universität Innsbruck, Austria

**Aart Middeldorp**
Department of Computer Science, Universiät Innsbruck, Austria
Future Value Creation Research Center, Nagoya University, Japan

## Abstract

In this paper we prove via a reduction from Hilbert's 10th problem that the problem whether the termination of a given rewrite system can be shown by a polynomial interpretation in the natural numbers is undecidable, even for rewrite systems that are incrementally polynomially terminating. We also prove that incremental polynomial termination is an undecidable property of terminating rewrite systems.

**2012 ACM Subject Classification** Theory of computation → Equational logic and rewriting; Theory of computation → Rewrite systems; Theory of computation → Computability

**Keywords and phrases** Term Rewriting, Polynomial Termination, Undecidability

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.27

## 1 Introduction

Proving termination of a rewrite system by using a polynomial interpretation over the natural numbers goes back to Lankford [10]. Two problems need to be addressed when using polynomial interpretations for proving termination, whether by hand or by a tool:

1. finding suitable polynomials for the function symbols,
2. showing that the induced order constraints on polynomials are valid.

Heuristics for the former problem are presented in [3, 18]. The latter problem amounts to ($\star$) proving $P(x_1, \ldots, x_n) > 0$ for all natural numbers $x_1, \ldots, x_n \in \mathbb{N}$, for polynomials $P \in \mathbb{Z}[x_1, \ldots, x_n]$. This is known to be undecidable, as an easy consequence of Hilbert's 10th Problem, see e.g., Zantema [18, Proposition 6.2.11]. However, from the undecidability of problem 2 it does not immediately follow that (dis)proving polynomial termination is undecidable, since a decision procedure for problem 1 may exist which only produces decidable instances for problem 2.

In this paper we show that this is not the case, by proving the undecidability of the existence of a direct termination proof by a polynomial interpretation in ℕ by a reduction from ($\star$). This result is not surprising, but we are not aware of a proof of undecidability in the literature, and the construction is not entirely obvious. We strengthen the undecidability result to rewrite systems that can be shown terminating by an incremental polynomial interpretation in ℕ, where rules are not oriented all at once, but in stages (called *lexicographic combinations* in [18, Section 6.2.4]). We further show that the existence of an incremental polynomial termination proof is undecidable for terminating rewrite systems.

In the next section we recall the definitions of (incremental) polynomial termination over ℕ. In Section 3 we present the variations of Hilbert's 10th problem that we use to obtain our undecidability results. The latter are presented in detail in the subsequent three sections. The undecidability result in Section 4 was first announced at the *International Workshop on Termination* in 2021 [14]. We conclude in Section 7 with suggestions for future work.

© Fabian Mitterwallner and Aart Middeldorp;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 27; pp. 27:1–27:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 2 Preliminaries

We assume familiarity with term rewriting [2], but recall the definition of (incremental) polynomial termination over $\mathbb{N}$. Given a signature $\mathcal{F}$, a well-founded monotone $\mathcal{F}$-algebra $(\mathcal{A}, >)$ consists of a non-empty $\mathcal{F}$-algebra $\mathcal{A} = (A, \{f_\mathcal{A}\}_{f \in \mathcal{F}})$ and a well-founded order $>$ on the carrier $A$ of $\mathcal{A}$ such that every algebra operation is strictly monotone in all its arguments, i.e., if $f \in \mathcal{F}$ has arity $n \geqslant 1$ then $f_\mathcal{A}(a_1, \ldots, a_i, \ldots, a_n) > f_\mathcal{A}(a_1, \ldots, b, \ldots, a_n)$ for all $a_1, \ldots, a_n, b \in A$ and $i \in \{1, \ldots, n\}$ with $a_i > b$. The induced order $>_\mathcal{A}$ on terms is a reduction order that ensures the termination of any compatible (i.e., $\ell >_\mathcal{A} r$ for all rewrite rules $\ell \to r$) term rewrite system (TRS for short) $\mathcal{R}$. We call $\mathcal{R}$ *polynomially terminating over* $\mathbb{N}$ if compatibility holds when the underlying algebra $\mathcal{A}$ is restricted to the set of natural numbers $\mathbb{N}$ with standard order $>_\mathbb{N}$ such that every $n$-ary function symbol $f$ is interpreted as a monotone polynomial $f_\mathbb{N}$ in $\mathbb{Z}[x_1, \ldots, x_n]$ with $f_\mathbb{N}(0, \ldots, 0) \geqslant 0$. The latter condition is needed for $f_\mathbb{N}$ to be well-defined over $\mathbb{N}$. We use $\mathbb{N}_+$ to denote $\mathbb{N} \setminus \{0\}$.

Whereas well-founded monotone algebras are complete for termination, polynomial termination gives rise to a much more restricted class of TRSs. For instance, Hofbauer and Lautemann [8] proved that polynomially terminating TRSs induce a double-exponential upper bound on the derivational complexity. Polynomial interpretations can be used in an incremental fashion, extending their termination proving power. The idea is that in a first step a polynomial interpretation is used that orients all rewrite rules of a given TRS $\mathcal{R}$ weakly and at least one rule strictly. After removing the rules that are strictly oriented, the process is repeated. (This is of course not specific to polynomial interpretations and more generally known as proving termination via relative termination [6, Chapter 3.2].) When no rule remains, the incremental termination proof succeeds. In this case, $\mathcal{R}$ is called *incremental polynomially terminating over* $\mathbb{N}$. The following example is from [18, Example 6.2.21].

▶ **Example 1.** Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{0} + y \to y \qquad \mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad \mathsf{0} \times y \to \mathsf{0} \qquad \mathsf{s}(x) \times y \to (x \times y) + y$$

The polynomial interpretation

$$\mathsf{0}_\mathbb{N} = 0 \qquad \mathsf{s}_\mathbb{N}(x) = x + 2 \qquad +_\mathbb{N}(x, y) = x + y + 2 \qquad \times_\mathbb{N}(x, y) = xy + 2x + 2y + 2$$

gives rise to the following order constraints on $\mathbb{N}$:

$$y + 2 > y \quad x + y + 4 = x + y + 4 \quad 2y + 2 > 0 \quad xy + 2x + 4y + 6 > xy + 2x + 3y + 4$$

So three of the four rules are oriented strictly. The exception is the rule $\mathsf{s}(x) + y \to \mathsf{s}(x + y)$, which is turned into an equality. Changing the interpretation to

$$\mathsf{s}_\mathbb{N}(x) = x + 1 \qquad\qquad +_\mathbb{N}(x, y) = 2x + y$$

orients this rule strictly. Hence $\mathcal{R}$ is incremental polynomially terminating over $\mathbb{N}$. With some effort, it can be shown that $\mathcal{R}$ is not polynomially terminating over $\mathbb{N}$. So $\mathcal{R}$ resides in the middle ring in Figure 1.

## 3 Hilbert's 10th Problem

In 1901 David Hilbert published a list of 23 mathematical problems, all of which were unsolved at the time [7]. The tenth problem on the list asked for a procedure to solve Diophantine equations.

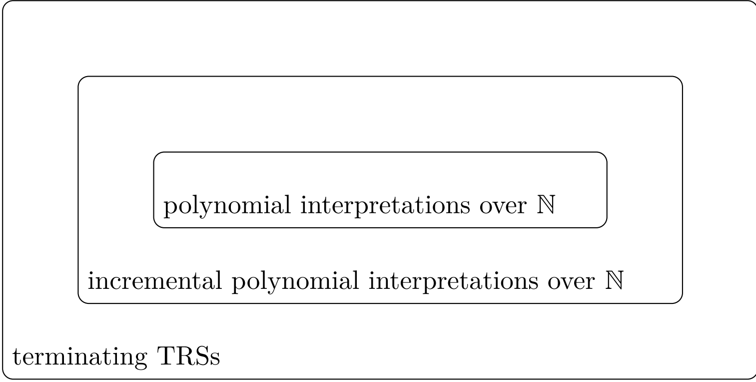



**Figure 1** Polynomial termination hierarchy.

▶ **Problem 2** (Hilbert 10)**.** *Given a polynomial* $P \in \mathbb{Z}[x_1, \ldots, x_n]$*, determine if there exists* $x_1, \ldots, x_n \in \mathbb{Z}$ *such that* $P(x_1, \ldots, x_n) = 0$.

In 1970 Yuri Matiyasevich showed that recursively enumerable sets are diophantine [12]. From this it follows that Hilbert's 10th problem is undecidable [4].

▶ **Theorem 3.** *Hilbert's 10th problem is undecidable.*

In this paper we use the following three variations of Hilbert's 10th problem, all of which are undecidable. Like Hilbert's 10th problem, (2) and (3) are semi-decidable, in other words the "yes" instances can be answered in finite time. Due to the universal quantification this is not the case for (1), which is co-semi-decidable, meaning that the "no" instances can be answered in finite time.

▶ **Theorem 4.** *The following decision problems are undecidable.*

**(1)** *instance:* *a polynomial* $P \in \mathbb{Z}[x_1, \ldots, x_n]$
*question:* $P(x_1, \ldots, x_n) > 0$ *for all* $x_1, \ldots, x_n \in \mathbb{N}$*?*

**(2)** *instance:* *a polynomial* $P \in \mathbb{Z}[x_1, \ldots, x_n]$
*question:* $P(x_1, \ldots, x_n) = 0$ *for some* $x_1, \ldots, x_n \in \mathbb{N}_+$*?*

**(3)** *instance:* *a polynomial* $P \in \mathbb{Z}[x_1, \ldots, x_n]$
*question:* $P(x_1, \ldots, x_n) \geqslant 0$ *for some* $x_1, \ldots, x_n \in \mathbb{N}_+$*?*

**Proof.** This follows by a reduction from Problem 2. We show this for (3). The other statements can be shown in a similar way (cf. [18, Proposition 6.2.11]). Assume there exists a procedure to solve (3) and let $P \in \mathbb{Z}[x_1, \ldots, x_n]$ be some polynomial. We can modify the original question of Hilbert's 10th problem as follows:

$$\begin{aligned}
&\exists\, x_1, \ldots, x_n \in \mathbb{Z}\ P(x_1, \ldots, x_n) = 0 \\
&\quad\iff \exists\, x_1, \ldots, x_n \in \mathbb{Z}\ \neg(P(x_1, \ldots, x_n)^2 > 0) \\
&\quad\iff \exists\, x_1, \ldots, x_n \in \mathbb{Z}\ \neg(-P(x_1, \ldots, x_n)^2 < 0) \\
&\quad\iff \exists\, x_1, \ldots, x_n \in \mathbb{Z}\ -P(x_1, \ldots, x_n)^2 \geqslant 0 \\
&\quad\iff \exists\, a_1, \ldots, a_n \in \{-1, 0, 1\}\ \exists\, y_1, \ldots, y_n \in \mathbb{N}_+\ -P(a_1 y_1, \ldots, a_n y_n)^2 \geqslant 0
\end{aligned}$$

For each tuple $\vec{a} = (a_1, \ldots, a_n) \in \{-1, 0, 1\}^n$, we construct the polynomial $Q_{\vec{a}}(y_1, \ldots, y_n) = -P(a_1 y_1, \ldots, a_n y_n)^2$. From our assumption "$\exists\, y_1, \ldots, y_n \in \mathbb{N}_+\ Q_{\vec{a}}(y_1, \ldots, y_n) \geqslant 0$" is decidable for all $\vec{a}$. Since there only exist finitely many such tuples, this proves decidability of Problem 2. This obviously contradicts Theorem 3 and hence (3) is undecidable. ◀

**Table 1** The TRS $\mathcal{R}$.

| | | | |
|---|---|---|---|
| $\mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{f}(x)))$ | (A) | $\mathsf{s}(\mathsf{s}(\mathsf{0})) \to \mathsf{q}(\mathsf{s}(\mathsf{0}))$ | (G) |
| $\mathsf{q}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{f}(\mathsf{q}(x)))$ | (B) | $\mathsf{s}(\mathsf{0}) \to \mathsf{q}(\mathsf{0})$ | (H) |
| $\mathsf{f}(x) \to \mathsf{a}(x, x)$ | (C) | $\mathsf{s}^5(\mathsf{0}) \to \mathsf{q}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$ | (I) |
| $\mathsf{s}(x) \to \mathsf{a}(\mathsf{0}, x)$ | (D) | $\mathsf{q}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) \to \mathsf{s}^3(\mathsf{0})$ | (J) |
| $\mathsf{s}(x) \to \mathsf{a}(x, \mathsf{0})$ | (E) | $\mathsf{s}(\mathsf{a}(x, x)) \to \mathsf{d}(x)$ | (K) |
| $\mathsf{a}(\mathsf{q}(x), \mathsf{f}(x)) \to \mathsf{q}(\mathsf{s}(x))$ | (F) | $\mathsf{s}(\mathsf{d}(x)) \to \mathsf{a}(x, x)$ | (L) |

$$\mathsf{s}(\mathsf{a}(\mathsf{q}(\mathsf{a}(x, y)), \mathsf{d}(\mathsf{a}(x, y)))) \to \mathsf{a}(\mathsf{a}(\mathsf{q}(x), \mathsf{q}(y)), \mathsf{d}(\mathsf{m}(x, y))) \quad \text{(M)}$$

$$\mathsf{s}(\mathsf{a}(\mathsf{a}(\mathsf{q}(x), \mathsf{q}(y)), \mathsf{d}(\mathsf{m}(x, y)))) \to \mathsf{a}(\mathsf{q}(\mathsf{a}(x, y)), \mathsf{d}(\mathsf{a}(x, y))) \quad \text{(N)}$$

## 4 Undecidability of Polynomial Termination

In this section we construct a family of TRSs $\mathcal{R}_P$ parameterized by polynomials $P \in \mathbb{Z}[x_1, \dots, x_n]$ such that $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$ if and only if $P(x_1, \dots, x_n) > 0$ for all $x_1, \dots, x_n \in \mathbb{N}$. The construction is based on techniques from [15], in which specific rewrite rules enforce the interpretations of certain function symbols. Our TRSs $\mathcal{R}_P$ consists of three parts: A fixed component $\mathcal{R}$, which is extended to $\mathcal{R}_k$ for some $k \in \mathbb{N}$ depending on the exponents in $P$, and a single rewrite rule that encodes the positiveness of $P$.

▶ **Definition 5.** *Given a polynomial $P \in \mathbb{Z}[x_1, \dots, x_n]$, the* TRS *$\mathcal{R}_P$ is defined over the signature consisting of a constant* $\mathsf{0}$*, unary function symbols* $\mathsf{s}$, $\mathsf{d}$, $\mathsf{f}$, $\mathsf{q}$, $\mathsf{p}_1, \dots, \mathsf{p}_k$*, and binary function symbols* $\mathsf{a}$ *and* $\mathsf{m}$*. Here $k$ is the highest degree of an indeterminate in $P$.*

To encode the positiveness of $P$ we need to constrain the possible interpretations of function symbols, in order to represent numbers, addition and multiplication. That is the purpose of the TRS $\mathcal{R}$, whose rules are presented in Table 1. It is a simplified and modified version of the TRS $\mathcal{R}_2$ in [15]. As will be shown later, this setup allows us to represent natural numbers as terms using the symbol $\mathsf{0}$ for the number 0 and $\mathsf{s}$ for the successor function. For the operations we have the binary symbols $\mathsf{a}$ for addition and $\mathsf{m}$ for multiplication. However, since multiplication is not strictly monotone on $\mathbb{N}$ we restrict the interpretation of $\mathsf{m}$ to $xy + x + y$, which suffices for the reduction. The remaining function symbols in $\mathcal{R}$ are not used to encode the positiveness of $P$, but are required for the construction to work. First we show that the mentioned interpretations prove termination of $\mathcal{R}$.

▶ **Lemma 6.** *The* TRS *$\mathcal{R}$ is polynomially terminating over $\mathbb{N}$.*

**Proof.** The well-founded algebra $(\mathbb{N}, >_{\mathbb{N}})$ with interpretations

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{a}_{\mathbb{N}}(x, y) = x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = x^2$$
$$\mathsf{d}_{\mathbb{N}}(x) = 2x \qquad \mathsf{f}_{\mathbb{N}}(x) = 4x + 6 \qquad \mathsf{m}_{\mathbb{N}}(x, y) = xy + x + y$$

is monotone and compatible with $\mathcal{R}$. Hence $\mathcal{R}$ is polynomially terminating. ◀

Note that this polynomial interpretation is found by the termination tool $\mathsf{T_TT_2}$ with the strategy `poly -direct -nl2 -ib 4 -ob 6`.

More importantly, to ensure termination in $(\mathbb{N}, >_{\mathbb{N}})$, the rewrite rules of $\mathcal{R}$ require that the interpretation of some of the function symbols is unique. The proof of the following lemma closely follows the reasoning in [15, Lemmata 4.4 and 5.2].

▶ **Lemma 7.** *Any monotone polynomial interpretation* $(\mathbb{N}, >_{\mathbb{N}})$ *compatible with* $\mathcal{R}$ *must interpret the function symbols* $\mathsf{0}$, $\mathsf{s}$, $\mathsf{d}$, $\mathsf{a}$, $\mathsf{m}$ *and* $\mathsf{q}$ *as follows:*

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{a}_{\mathbb{N}}(x, y) = x + y$$
$$\mathsf{d}_{\mathbb{N}}(x) = 2x \qquad \mathsf{m}_{\mathbb{N}}(x, y) = xy + x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = x^2$$

**Proof.** Compatibility with (A) implies $\deg(\mathsf{f}_{\mathbb{N}}) \cdot \deg(\mathsf{s}_{\mathbb{N}}) \geqslant \deg(\mathsf{s}_{\mathbb{N}})^2 \cdot \deg(\mathsf{f}_{\mathbb{N}})$. This is only possible if $\deg(\mathsf{s}_{\mathbb{N}}) \leqslant 1$. Together with the strict monotonicity of $\mathsf{s}_{\mathbb{N}}$ we obtain $\deg(\mathsf{s}_{\mathbb{N}}) = 1$. Hence $\mathsf{s}$ must be interpreted by a linear polynomial: $\mathsf{s}_{\mathbb{N}}(x) = s_1x + s_0$ with $s_1 \geqslant 1$ and $s_0 \geqslant 0$. The same reasoning applied to (B) yields $\mathsf{f}_{\mathbb{N}}(x) = f_1x + f_0$ for some $f_1 \geqslant 1$ and $f_0 \geqslant 0$. The compatibility constraint imposed by rule (A) further gives rise to the inequality

$$f_1s_1x + f_1s_0 + f_0 > f_1s_1^2x + f_0s_1^2 + s_1s_0 + s_0 \tag{1}$$

for all $x \in \mathbb{N}$. Since $s_1 \geqslant 1$ and $f_1 \geqslant 1$, this only holds if $s_1 = 1$. Simplifying (1) we obtain $f_1s_0 > 2s_0$, which implies $s_0 > 0$ and $f_1 > 2$. If $\mathsf{q}_{\mathbb{N}}$ were linear, the same reasoning could be applied to (B) resulting in $f_1 = 1$, contradicting $f_1 > 2$. Hence $\mathsf{q}_{\mathbb{N}}$ is at least quadratic.

Next we turn our attention to the rewrite rules (C) – (F). Because $\mathsf{f}_{\mathbb{N}}$ is linear, compatibility with (C) and strict monotonicity of $\mathsf{a}_{\mathbb{N}}$ ensures $\deg(\mathsf{a}_{\mathbb{N}}) = 1$. Hence, $\mathsf{a}_{\mathbb{N}} = a_2x + a_1y + a_0$ with $a_2 \geqslant 1$, $a_1 \geqslant 1$ and $a_0 \geqslant 0$. From compatibility with rules (D) and (E) we obtain $a_1 = 1$ and $a_2 = 1$. Using the current shapes of $\mathsf{a}_{\mathbb{N}}$, $\mathsf{f}_{\mathbb{N}}$ and $\mathsf{s}_{\mathbb{N}}$, compatibility with rule (F) yields the inequality $\mathsf{f}_{\mathbb{N}}(x) + a_0 > \mathsf{q}_{\mathbb{N}}(x + s_0) - \mathsf{q}_{\mathbb{N}}(x)$ for all $x \in \mathbb{N}$. This can only be the case if $\deg(\mathsf{f}_{\mathbb{N}}(x) + a_0) \geqslant \deg(\mathsf{q}_{\mathbb{N}}(x + s_0) - \mathsf{q}_{\mathbb{N}}(x))$, which in turn simplifies to $1 \geqslant \deg(\mathsf{q}_{\mathbb{N}}(x)) - 1$. Hence $\mathsf{q}_{\mathbb{N}}(x) = q_2x^2 + q_1x + q_0$ with $q_2 \geqslant 1$. From monotonicity we also have $\mathsf{q}_{\mathbb{N}}(1) > \mathsf{q}_{\mathbb{N}}(0)$, which leads to $q_2 + q_1 \geqslant 1$.

To further constrain $\mathsf{s}_{\mathbb{N}}$ we consider the rewrite rule (G). The compatibility constraint gives rise to

$$\begin{aligned}
\mathsf{0}_{\mathbb{N}} + 2s_0 &> q_2(\mathsf{0}_{\mathbb{N}} + s_0)^2 + q_1(\mathsf{0}_{\mathbb{N}} + s_0) + q_0 \\
&= q_2\mathsf{0}_{\mathbb{N}}^2 + q_2s_0^2 + \mathsf{0}_{\mathbb{N}}(2q_2s_0 + q_1) + q_1s_0 + q_0 \\
&\geqslant q_2s_0^2 + \mathsf{0}_{\mathbb{N}} + (1 - q_2)s_0 && (q_2 + q_1 \geqslant 1 \text{ and } q_0, q_2, s_0 \geqslant 1) \\
&= q_2s_0(s_0 - 1) + \mathsf{0}_{\mathbb{N}} + s_0 \geqslant s_0^2 + \mathsf{0}_{\mathbb{N}} && (s_0 \geqslant 1)
\end{aligned}$$

Hence the inequality $2s_0 > s_0^2$ holds, which is only true if $s_0 = 1$. Therefore $\mathsf{s}_{\mathbb{N}}(x) = x + 1$. Compatibility with (D) now amounts to $x + 1 > \mathsf{0}_{\mathbb{N}} + x + a_0$, which implies $\mathsf{0}_{\mathbb{N}} = a_0 = 0$. At this point we have uniquely constrained $\mathsf{0}_{\mathbb{N}}$, $\mathsf{s}_{\mathbb{N}}$ and $\mathsf{a}_{\mathbb{N}}$. To fully constrain $\mathsf{q}_{\mathbb{N}}$ we turn to (H), which implies $q_0 = 0$, the rule (G), which together with monotonicity implies $2 > \mathsf{q}_{\mathbb{N}}(1) > 0$ and thus $\mathsf{q}_{\mathbb{N}}(1) = q_2 + q_1 = 1$, and the rules (I) and (J), which imply $5 > \mathsf{q}_{\mathbb{N}}(2) > 3$ and thus $\mathsf{q}_{\mathbb{N}}(2) = 4q_2 + 2q_1 = 4$. Consequently, $q_2 = 1$ and $q_1 = 0$. Hence $\mathsf{q}_{\mathbb{N}}(x) = x^2$. Compatibility with the rules (K) and (L) yields $x + x + 1 > \mathsf{d}_{\mathbb{N}}(x)$ and $\mathsf{d}_{\mathbb{N}}(x) + 1 > x + x$ which imply $\mathsf{d}_{\mathbb{N}}(x) = 2x$. Finally, compatibility with the rules (M) and (N) amounts to $(x + y)^2 + 2x + 2y + 1 > x^2 + y^2 + 2\mathsf{m}_{\mathbb{N}}(x, y) \geqslant (x + y)^2 + 2x + 2y$, which uniquely determines $\mathsf{m}_{\mathbb{N}}(x, y) = xy + x + y$. ◀

Using the previously fixed interpretations we can easily restrict the interpretations of any new symbols. By adding the two rules

$$\mathsf{s}(t) \to u \qquad\qquad \mathsf{s}(u) \to t$$

for some terms $t$ and $u$, we enforce an equality constraint on the interpretations of $t$ and $u$, assuming the system remains polynomially terminating.

To represent the exponents in the polynomial $P$ we use the symbols $\mathsf{p}_i$ for $1 \leqslant i \leqslant k$, where $k$ is the maximal exponent in $P$. To fix $\mathsf{p}_{i\mathbb{N}}(x) = x^i$, we add two rules per symbol, according to the following definition.

▶ **Definition 8.** *We define a family of* TRS*s* $(\mathcal{R}_k)_{k\geqslant 0}$ *as follows:*

$$\mathcal{R}_0 = \mathcal{R}$$
$$\mathcal{R}_1 = \mathcal{R}_0 \cup \{\mathsf{s}(\mathsf{p}_1(x)) \to x,\ \mathsf{s}(x) \to \mathsf{p}_1(x)\}$$
$$\mathcal{R}_{k+1} = \mathcal{R}_k \cup \left\{ \begin{aligned} &\mathsf{s}(\mathsf{a}(\mathsf{p}_{k+1}(x), \mathsf{a}(x, \mathsf{p}_k(x)))) \to \mathsf{m}(x, \mathsf{p}_k(x)) \\ &\qquad\qquad \mathsf{s}(\mathsf{m}(x, \mathsf{p}_k(x))) \to \mathsf{a}(\mathsf{p}_{k+1}(x), \mathsf{a}(x, \mathsf{p}_k(x))) \end{aligned} \right\}$$

▶ **Lemma 9.** *For any $k \geqslant 0$, the* TRS *$\mathcal{R}_k$ is polynomially terminating over* $\mathbb{N}$ *if and only if* $\mathsf{p}_{i\mathbb{N}}(x) = x^i$ *for all* $1 \leqslant i \leqslant k$.

**Proof.** From Lemma 6 we know that $\mathcal{R}$ is polynomially terminating and the interpretations are unique due to Lemma 7. Hence the lemma holds for $\mathcal{R}_0$. For $k \geqslant 1$, the *if* direction holds, since the interpretations $\mathsf{p}_{i\mathbb{N}}$ are monotone and the polynomial interpretation is compatible with $\mathcal{R}_k$:

$$x + 1 > x \qquad\qquad x + 1 > x$$

for $\mathcal{R}_1 \setminus \mathcal{R}_0$ and

$$x^k + x + x^{k-1} + 1 > xx^{k-1} + x + x^{k-1} \qquad xx^{k-1} + x + x^{k-1} + 1 > xx^k + x + x^{k-1}$$

for $\mathcal{R}_k \setminus \mathcal{R}_{k-1}$. For the *only if* direction we show that compatibility with the additional rules implies $\mathsf{p}_{i\mathbb{N}}(x) = x^i$ for all $1 \leqslant i \leqslant k$. This is done by induction on $k$. For $k = 1$ the two rules in $\mathcal{R}_1 \setminus \mathcal{R}$ enforce $\mathsf{p}_{i\mathbb{N}}(x) + 1 > x$ and $x + 1 > \mathsf{p}_{i\mathbb{N}}(x)$. Hence $\mathsf{p}_{i\mathbb{N}}(x) = x$. For $k > 1$ the rules in $\mathcal{R}_k \setminus \mathcal{R}_{k-1}$ enforce $\mathsf{p}_{k\mathbb{N}}(x) = x \cdot \mathsf{p}_{k-1\mathbb{N}}(x)$ by the same reasoning. From the induction hypothesis we obtain $\mathsf{p}_{k-1\mathbb{N}}(x) = x^{k-1}$ and hence $\mathsf{p}_{k\mathbb{N}} = x^k$ as desired. ◀

The fixed interpretations can now be used to construct arbitrary polynomials. Since non-monotone operations, such as subtraction (negative coefficients) and multiplication, cannot serve as interpretations for function symbols, we model these using the difference of two terms. In the following we write $[t]_\mathbb{N}$ for the polynomial that is the interpretation of the term $t$, according to the interpretations stated in Lemmata 7 and 9.

▶ **Lemma 10.** *For any monomial $M = c\,x_1^{i_1} \cdots x_m^{i_m}$ with $i_1, \ldots, i_m > 0$ and $c \neq 0$ there exist terms $\ell_M$ and $r_M$ over the signature of $\mathcal{R}_{\mathsf{max}(0,i_1,\ldots,i_m)}$, such that $M = [\ell_M]_\mathbb{N} - [r_M]_\mathbb{N}$ and $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(r_M)$.*

**Proof.** First we assume the coefficient $c$ is positive. We construct $\ell_M$ and $r_M$ by induction on $m$. If $m = 0$ then $M = c$ and we take $\ell_M = \mathsf{s}^c(\mathsf{0})$ and $r_M = \mathsf{0}$. We trivially have $\mathcal{V}\mathsf{ar}(\ell_M) = \varnothing = \mathcal{V}\mathsf{ar}(r_M)$ and $[\ell_M]_\mathbb{N} - [r_M]_\mathbb{N} = c - 0 = M$. For $m > 0$ we have $M = M' x_m^{i_m}$ with $M' = c x_1^{i_1} \cdots x_{m-1}^{i_{m-1}}$. The induction hypothesis yields terms $\ell_{M'}$ and $r_{M'}$ with $M' = [\ell_{M'}]_\mathbb{N} - [r_{M'}]_\mathbb{N}$ and $\mathcal{V}\mathsf{ar}(\ell_{M'}) = \mathcal{V}\mathsf{ar}(r_{M'})$. Hence

$$\begin{aligned} M &= M' x_m^{i_m} = [\ell_{M'}]_\mathbb{N} x_m^{i_m} - [r_{M'}]_\mathbb{N} x_m^{i_m} \\ &= (\mathsf{m}_\mathbb{N}([\ell_{M'}]_\mathbb{N}, x_m^{i_m}) - [\ell_{M'}]_\mathbb{N} - x_m^{i_m}) - (\mathsf{m}_\mathbb{N}([r_{M'}]_\mathbb{N}, x_m^{i_m}) - [r_{M'}]_\mathbb{N} - x_m^{i_m}) \\ &= (\mathsf{m}_\mathbb{N}([\ell_{M'}]_\mathbb{N}, \mathsf{p}_{i_m\mathbb{N}}(x_m)) + [r_{M'}]_\mathbb{N}) - (\mathsf{m}_\mathbb{N}([r_{M'}]_\mathbb{N}, \mathsf{p}_{i_m\mathbb{N}}(x_m)) + [\ell_{M'}]_\mathbb{N}) \end{aligned}$$

and thus we can take $\ell_M = \mathsf{a}(\mathsf{m}(\ell_{M'}, \mathsf{p}_{i_m}(x_m)), r_{M'})$ and $r_M = \mathsf{a}(\mathsf{m}(r_{M'}, \mathsf{p}_{i_m}(x_m)), \ell_{M'})$. Note that $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(\ell_{M'}) \cup \{x_m\} \cup \mathcal{V}\mathsf{ar}(r_{M'}) = \mathcal{V}\mathsf{ar}(r_M)$.

If $c < 0$ then we take $\ell_M = r_{-M}$ and $r_M = \ell_{-M}$. We obviously have $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(r_{-M}) = \mathcal{V}\mathsf{ar}(\ell_{-M}) = \mathcal{V}\mathsf{ar}(r_M)$. Moreover,

$$M = -(-M) = -([\ell_{-M}]_{\mathbb{N}} - [r_{-M}]_{\mathbb{N}}) = -([r_M]_{\mathbb{N}} - [\ell_M]_{\mathbb{N}}) = [\ell_M]_{\mathbb{N}} - [r_M]_{\mathbb{N}}$$ ◀

▶ **Definition 11.** *Let $P = M_1 + \cdots + M_{l-1} + M_l \in \mathbb{Z}[x_1, \ldots, x_n]$ be a sum of monomials. We define $\ell_P = \mathsf{a}(\ell_{M_1}, \cdots \mathsf{a}(\ell_{M_{l-1}}, \ell_{M_l}) \cdots)$ and $r_P = \mathsf{a}(r_{M_1}, \cdots \mathsf{a}(r_{M_{l-1}}, r_{M_l}) \cdots)$. Moreover, $\ell_0 = r_0 = \mathsf{0}$. We define the* TRS *$\mathcal{R}_P$ as the extension of $\mathcal{R}_k$ with the single rule $\ell_P \to r_P$. Here $k$ is the maximal exponent occurring in $P$.*

Note that the rewrite rule $\ell_P \to r_P$ in $\mathcal{R}_P$ is well-defined; $\ell_P$ is not a variable and $\mathcal{V}\mathsf{ar}(\ell_P) = \mathcal{V}\mathsf{ar}(r_P)$ as a consequence of Lemma 10.

▶ **Example 12.** The polynomial $P = 2x^2y - xy + 3$ is first split into its monomials $M_1 = 2x^2y$, $M_2 = -xy$ and $M_3 = 3$. Hence we obtain the TRS $\mathcal{R}_{P_1} = \mathcal{R}_2 \cup \{\mathsf{a}(\ell_{M_1}, \mathsf{a}(\ell_{M_2}, \ell_{M_3})) \to \mathsf{a}(r_{M_1}, \mathsf{a}(r_{M_2}, r_{M_3}))\}$, where

$$\ell_{M_1} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}^2(\mathsf{0}), \mathsf{p}_2(x)), \mathsf{0})}_{\ell_{2x^2}}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_2(x)), \mathsf{s}^2(\mathsf{0}))}_{r_{2x^2}})$$

$$r_{M_1} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_2(x)), \mathsf{s}^2(\mathsf{0}))}_{r_{2x^2}}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}^2(\mathsf{0}), \mathsf{p}_2(x)), \mathsf{0})}_{\ell_{2x^2}})$$

$$\ell_{M_2} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))}_{r_x}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})}_{\ell_x})$$

$$r_{M_2} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})}_{\ell_x}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))}_{r_x})$$

$$\ell_{M_3} = \mathsf{s}^3(\mathsf{0}) \qquad r_{M_3} = \mathsf{0}$$

Note that in the terms $\ell_{M_2}$ and $r_{M_2}$ the $\ell$ and $r$ of the recursive call are switched since $M_2$ has a negative coefficient.

▶ **Theorem 13.** *For any polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$, the* TRS *$\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$ if and only if $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$.*

**Proof.** First suppose $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$. So there exists a monotone polynomial interpretation in $(\mathbb{N}, >)$ that orients the rules of $\mathcal{R}_P$ from left to right. Let $k$ be the maximum exponent in $P$. From Lemma 7 and Lemma 9 we infer that the interpretations of the function symbols $\mathsf{0}$, $\mathsf{s}$, $\mathsf{a}$, $\mathsf{m}$, and $\mathsf{p}_i$ for $1 \leqslant i \leqslant k$ are fixed such that, according to Lemma 10, $P = [\ell_P]_{\mathbb{N}} - [r_P]_{\mathbb{N}}$. Since the rule $\ell_P \to r_P$ belongs to $\mathcal{R}_P$, $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$ by compatibility.

For the if direction, we assume that $P \in \mathbb{Z}[x_1, \ldots, x_n]$ satisfies $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$. By construction of $\ell_P \to r_P$ and Lemma 10, the interpretations in Lemma 7 and Lemma 9 orient the rule $\ell_P \to r_P$ from left to right. The same holds for rules $\mathcal{R}_n$. Hence $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$. ◀

▶ **Corollary 14.** *It is undecidable whether a* TRS *is polynomially terminating over $\mathbb{N}$.*

Since the proof reduces polynomial termination to (1) from Theorem 4, which is not semi-decidable, the same holds for polynomial termination.

The construction of $\mathcal{R}_P$ may produce non-terminating systems. Take for example the polynomial $P_1 = -1$. The resulting TRS $\mathcal{R}_{P_1} = \mathcal{R} \cup \{\mathsf{0} \to \mathsf{s}(\mathsf{0})\}$ is obviously not terminating. Hence the question remains whether polynomial termination over $\mathbb{N}$ is undecidable for terminating TRSs. In the next section we show that this is indeed the case.

Another question is whether incremental polynomial termination over $\mathbb{N}$, where we take the lexicographic extension of the order induced by the polynomial interpretations, is decidable. The construction of $\mathcal{R}_P$ cannot be used to answer this question. Consider for instance the polynomial $P_2 = x$. We obtain $\ell_{P_2} = \mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})$ and $r_{P_2} = \mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))$. As a result, the TRS $\mathcal{R}_{P_2} = \mathcal{R}_1 \cup \{\ell_{P_2} \to r_{P_2}\}$ is not polynomially terminating over $\mathbb{N}$ since $[\ell_{P_2}]_\mathbb{N} = 2x + 1 \not> x + 1 = [r_{P_2}]_\mathbb{N}$ for $x = 0$. However, if we take a second interpretation over $\mathbb{N}$ where the interpretation of $\mathsf{m}$ is changed to $\mathsf{m}_\mathbb{N}(x, y) = 2x + y$, then $[\ell_{P_2}]_\mathbb{N} = x + 2 > x + 1 = [r_{P_2}]_\mathbb{N}$ for all $x \in \mathbb{N}$. Hence $\mathcal{R}_{P_2}$ is incremental polynomially terminating over $\mathbb{N}$. In Section 6 we provide a different construction which permits to answer the question about incremental polynomial termination over $\mathbb{N}$

## 5 Polynomial Termination of Terminating Rewrite Systems

In the reduction in the previous section indeterminates in the input polynomial $P$ are modeled as variables in the rewrite rule $\ell_P \to r_P$. In this and the next section we model indeterminates as unary function symbols. The following example illustrates how we intend to model the indeterminates as coefficients of the interpretation of the associated function symbol.

► **Example 15.** Suppose the interpretations of the function symbols $\mathsf{0}$, $\mathsf{s}$, and $\mathsf{a}$ are already fixed to 0, the successor function, and addition. Moreover, let $\mathsf{f}$ be a unary function symbol whose interpretation is linear without any upper bound on the coefficients. The rewrite rules

$$\mathsf{s}(\mathsf{0}) \to \mathsf{X}(\mathsf{0}) \qquad \mathsf{s}(\mathsf{0}) \to \mathsf{Y}(\mathsf{0}) \qquad \mathsf{f}(x) \to \mathsf{X}(x) \qquad \mathsf{f}(x) \to \mathsf{Y}(x)$$

constrain the interpretations of the unary function symbols $\mathsf{X}$ and $\mathsf{Y}$ to homogeneous linear polynomials: $\mathsf{X}_\mathbb{N}(x) = cx$ and $\mathsf{Y}_\mathbb{N}(x) = dx$, where $c, d > 0$. The claim that the polynomial $P(x, y) = x^2 + xy - x - 3$ has a root in $\mathbb{N}_+$ is equivalent to the claim that the rules

$$\mathsf{s}(\mathsf{a}(\mathsf{X}(\mathsf{s}(\mathsf{0})), \mathsf{s}^3(\mathsf{0}))) \to \mathsf{a}(\mathsf{X}(\mathsf{X}(\mathsf{s}(\mathsf{0}))), \mathsf{X}(\mathsf{Y}(\mathsf{s}(\mathsf{0}))))$$
$$\mathsf{s}(\mathsf{a}(\mathsf{X}(\mathsf{X}(\mathsf{s}(\mathsf{0}))), \mathsf{X}(\mathsf{Y}(\mathsf{s}(\mathsf{0}))))) \to \mathsf{a}(\mathsf{X}(\mathsf{s}(\mathsf{0})), \mathsf{s}^3(\mathsf{0}))$$

can be oriented by a polynomial interpretation, assuming the interpretations are constrained as above. To see this we look at the induced compatibility constraint of the two rules:

$$c + 3 = c^2 + cd$$

After some rearranging $c, d \in \mathbb{N}_+$ take the place of $x$ and $y$ in the polynomial. Hence this equation has a solution if and only if the polynomial has a root in the positive natural numbers.

Note that natural numbers and addition are still modeled using the symbols $\mathsf{0}$, $\mathsf{s}$ and $\mathsf{a}$, however multiplication of indeterminates (and a single coefficient) are now modeled using function composition. For example $2xy$ becomes $\mathsf{Y}(\mathsf{X}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))$. To make this possible we constrain the possible interpretations of these symbols using the TRS $\mathcal{C}_P$.

**Table 2** The TRS $\mathcal{C}$.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{f}(x)))$ | (A) | $\mathsf{f}(x) \to \mathsf{a}(x,x)$ | (D) | $\mathsf{s}(\mathsf{s}(\mathsf{0})) \to \mathsf{q}(\mathsf{s}(\mathsf{0}))$ | (G) |
| $\mathsf{q}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{f}(\mathsf{q}(x)))$ | (B) | $\mathsf{s}(x) \to \mathsf{a}(\mathsf{0},x)$ | (E) | $\mathsf{s}(\mathsf{A}(x)) \to \mathsf{B}(x)$ | (L) |
| $\mathsf{a}(\mathsf{q}(x),\mathsf{f}(x)) \to \mathsf{q}(\mathsf{s}(x))$ | (C) | $\mathsf{s}(x) \to \mathsf{a}(x,\mathsf{0})$ | (F) | $\mathsf{s}(\mathsf{B}(x)) \to \mathsf{A}(x)$ | (M) |

▶ **Definition 16.** *Given a polynomial $P \in \mathbb{Z}[x_1,\dots,x_n]$, the TRS $\mathcal{C}_P$ is defined over the signature consisting of a constant $\mathsf{0}$, unary function symbols $\mathsf{s}$, $\mathsf{f}$, $\mathsf{q}$, $\mathsf{X}_1,\dots,\mathsf{X}_n$, $\mathsf{A}$, $\mathsf{B}$, and a binary function symbol $\mathsf{a}$. It contains the rewrite rules presented in Table 2, which we denote by $\mathcal{C}$, as well as the $2n$ rules*

$$\mathsf{s}(\mathsf{0}) \to \mathsf{X}_i(\mathsf{0}) \quad (\mathrm{H}_i) \qquad\qquad \mathsf{f}(x) \to \mathsf{X}_i(x) \quad (\mathrm{I}_i)$$

*for all $1 \leqslant i \leqslant n$, which we denote by $\mathcal{X}_n$.*

The function symbols $\mathsf{A}$ and $\mathsf{B}$ in the rules (L) and (M), are not needed for modeling $P$ or for constraining the interpretations of the other symbol, but will be used to prove incremental polynomial termination of the TRS later.

▶ **Lemma 17.** *The TRS $\mathcal{C} \cup \mathcal{X}_n$ is polynomially terminating over $\mathbb{N}$, for any $n \geqslant 0$.*

**Proof.** The interpretations

$$\mathsf{0}_{\mathbb{N}} = 0 \quad \mathsf{s}_{\mathbb{N}}(x) = x+1 \quad \mathsf{a}_{\mathbb{N}}(x,y) = x+y \quad \mathsf{q}_{\mathbb{N}}(x) = x^2$$
$$\mathsf{A}_{\mathbb{N}}(x) = x \quad \mathsf{B}_{\mathbb{N}}(x) = x \quad \mathsf{f}_{\mathbb{N}}(x) = fx+f+2 \quad \mathsf{X}_{i\mathbb{N}}(x) = c_i x \quad \text{for } 1 \leqslant i \leqslant n$$

for any $c_1,\dots,c_n > 0$ and with $f = \max(3,c_1,\dots,c_n)$ are compatible with the rules in $\mathcal{C} \cup \mathcal{X}_n$. For the rules in $\mathcal{C}$ this can be seen as follows:

$$[\mathsf{f}(\mathsf{s}(x))]_{\mathbb{N}} = fx+2f+2 > fx+f+4 = [\mathsf{s}(\mathsf{s}(\mathsf{f}(x)))]_{\mathbb{N}} \quad (\mathrm{A})$$
$$[\mathsf{q}(\mathsf{f}(x))]_{\mathbb{N}} = f^2x^2+2fx(f+2)+f^2+4f+4 > f^2x^2+f^2+3f+2 = [\mathsf{f}(\mathsf{f}(\mathsf{q}(x)))]_{\mathbb{N}} \quad (\mathrm{B})$$
$$[\mathsf{a}(\mathsf{q}(x),\mathsf{f}(x))]_{\mathbb{N}} = x^2+fx+f+2 > x^2+2x+1 = [\mathsf{q}(\mathsf{s}(x))]_{\mathbb{N}} \quad (\mathrm{C})$$
$$[\mathsf{f}(x)]_{\mathbb{N}} = fx+f+2 > 2x = [\mathsf{a}(x,x)]_{\mathbb{N}} \quad (\mathrm{D})$$
$$[\mathsf{s}(x)]_{\mathbb{N}} = x+1 > x = [\mathsf{a}(\mathsf{0},x)]_{\mathbb{N}} \quad (\mathrm{E})$$
$$[\mathsf{s}(x)]_{\mathbb{N}} = x+1 > x = [\mathsf{a}(x,\mathsf{0})]_{\mathbb{N}} \quad (\mathrm{F})$$
$$[\mathsf{s}(\mathsf{s}(\mathsf{0}))]_{\mathbb{N}} = 2 > 1 = [\mathsf{q}(\mathsf{s}(\mathsf{0}))]_{\mathbb{N}} \quad (\mathrm{G})$$
$$[\mathsf{s}(\mathsf{A}(x))]_{\mathbb{N}} = x+1 > x = [\mathsf{B}(x)]_{\mathbb{N}} \quad (\mathrm{L})$$
$$[\mathsf{s}(\mathsf{B}(x))]_{\mathbb{N}} = x+1 > x = [\mathsf{A}(x)]_{\mathbb{N}} \quad (\mathrm{M})$$

For the rules in $\mathcal{X}_n$ we have

$$[\mathsf{s}(\mathsf{0})]_{\mathbb{N}} = 1 > 0 = [\mathsf{X}_i(\mathsf{0})]_{\mathbb{N}} \quad (\mathrm{H}_i)$$
$$[\mathsf{f}(x)]_{\mathbb{N}} = fx+f+2 > c_i x = [\mathsf{X}_i(x)]_{\mathbb{N}} \quad (\mathrm{I}_i)$$

◀

Before we can formally define the two ground rules that model "$P(x_1,\dots,x_n) = 0$ for some $x_1,\dots,x_n \in \mathbb{N}_+$," we need a preliminary definition which associates terms with polynomials.

▶ **Definition 18.** *Given a polynomial*

$$P = \sum_{i=1}^{m} M_i + \sum_{j=1}^{k} N_j \in \mathbb{Z}[x_1, \dots, x_n]$$

*such that the monomials $M_1, \dots, M_m$ have positive and the monomials $N_1, \dots, N_k$ have negative coefficients, we define $P_+ = M_1 + \cdots + M_m$ and $P_- = -(N_1 + \cdots + N_k)$. Given a monomial $M \in \mathbb{Z}[x_1, \dots, x_n]$ with positive coefficient, we define the term $t_M$ inductively as follows:*

$$t_M = \begin{cases} \mathsf{s}^c(\mathsf{0}) & \text{if } M = c \in \mathbb{N}_+ \\ \mathsf{X}^i(t_{M'}) & \text{if } M = M'x^i \end{cases}$$

*Given a polynomial $P = M_1 + \cdots + M_l \in \mathbb{Z}[x_1, \dots, x_n]$ with positive coefficients, we define the term $t_P$ inductively as follows:*

$$t_P = \begin{cases} \mathsf{0} & \text{if } l = 0 \\ t_{M_1} & \text{if } l = 1 \\ \mathsf{a}(t_{M_1}, t_{P-M_1}) & \text{otherwise} \end{cases}$$

▶ **Example 19.** For the polynomial $P(x, y) = x^3 - 2xy^2 + 3y - 2$ we obtain

$$t_{P_+} = \mathsf{a}(\mathsf{X}(\mathsf{X}(\mathsf{X}(\mathsf{s}(\mathsf{0})))), \mathsf{Y}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \qquad t_{P_-} = \mathsf{a}(\mathsf{Y}(\mathsf{Y}(\mathsf{X}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))), \mathsf{s}(\mathsf{s}(\mathsf{0})))$$

▶ **Definition 20.** *The* TRS $\mathcal{C}_P$ *is the extension of* $\mathcal{C} \cup \mathcal{X}_n$ *with the two ground rules*

$$\mathsf{A}(\mathsf{s}(t_{P_+})) \to \mathsf{B}(t_{P_-}) \quad \text{(J)} \qquad \mathsf{A}(\mathsf{s}(t_{P_-})) \to \mathsf{B}(t_{P_+}) \quad \text{(K)}$$

▶ **Theorem 21.** *For any polynomial $P \in \mathbb{Z}[x_1, \dots, x_n]$, the* TRS $\mathcal{C}_P$ *is polynomially terminating over ℕ if and only if $P(x_1, \dots, x_n) = 0$ for some $x_1, \dots, x_n \in \mathbb{N}_+$. Moreover, $\mathcal{C}_P$ is incremental polynomially terminating over ℕ.*

**Proof.** First suppose that $\mathcal{C}_P$ is polynomially terminating over ℕ. By the same reasoning as in the proof of Lemma 7, compatibility with the rules (A) – (G) in Table 2 ensures

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{a}_{\mathbb{N}}(x, y) = x + y$$

Moreover, the interpretation of $\mathsf{f}$ must be linear, i.e., $\mathsf{f}_{\mathbb{N}}(x) = f_1 x + f_0$, and $f_0 > f_1 + 1$. The rules (L) and (M) mandate $\mathsf{A}_{\mathbb{N}}(x) = \mathsf{B}_{\mathbb{N}}(x)$ for all $x \in \mathbb{N}$. Importantly this also means $\mathsf{A}_{\mathbb{N}}(x) = \mathsf{B}_{\mathbb{N}}(y)$ implies $x = y$ for all $x, y \in \mathbb{N}$ due to monotonicity. The rules in $\mathcal{R}_n$ constrain the interpretations of the symbols $\mathsf{X}_1, \dots, \mathsf{X}_n$ to $\mathsf{X}_{i\mathbb{N}}(x) = c_i x$ with arbitrary values $c_1, \dots, c_n \in \mathbb{N}_+$. Hence $[t_M]_{\mathbb{N}} = c\, c_1^{i_1} \cdots c_n^{i_n}$ for a monomial $M = c\, x_1^{i_1} \cdots x_n^{i_n}$ with $c > 0$, and thus also $[t_Q]_{\mathbb{N}} = Q(c_1, \dots, c_n)$ for a polynomial with positive coefficients. Consequently, the two rules in Definition 20 induce the constraint

$$P_+(c_1, \dots, c_n) = P_-(c_1, \dots, c_n)$$

This constraint is satisfiable if and only if the polynomial $P_+(x_1, \dots, x_n) - P_-(x_1, \dots, x_n) = P(x_1, \dots, x_n)$ has a root in $\mathbb{N}_+$. Conversely, suppose $P(a_1, \dots, a_n) = 0$ for some $a_1, \dots, a_n \in \mathbb{N}_+$. From Lemma 17 we obtain that $\mathcal{C} \cup \mathcal{X}_n$ is polynomially terminating over ℕ. According

to the proof of Lemma 17, we are free to choose the (positive) coefficients $c_1, \ldots, c_n$ of $\mathsf{X}_{1\mathbb{N}}, \ldots, \mathsf{X}_{n\mathbb{N}}$. By taking $c_i = a_i$ for $1 \leqslant i \leqslant n$, we ensure $[t_{P_+}]_\mathbb{N} = [t_{P_-}]_\mathbb{N}$ and hence $\mathcal{C}_P$ is polynomially terminating over $\mathbb{N}$.

For the second statement we start with the interpretations

$$\begin{array}{llll} \mathsf{0}_\mathbb{N} = 0 & \mathsf{s}_\mathbb{N}(x) = x & \mathsf{a}_\mathbb{N}(x,y) = x + y & \mathsf{q}_\mathbb{N}(x) = 2x^2 + x \\ \mathsf{f}_\mathbb{N}(x) = 2x + 2 & \mathsf{A}_\mathbb{N}(x) = x & \mathsf{B}_\mathbb{N}(x) = x & \mathsf{X}_{i\mathbb{N}}(x) = x \quad \text{for } 1 \leqslant i \leqslant n \end{array}$$

that strictly orients (B), (C), (D) and (I$_i$):

$$[\mathsf{q}(\mathsf{f}(x))]_\mathbb{N} = 8x^2 + 18x + 10 > 8x^2 + 4x + 6 = [\mathsf{f}(\mathsf{f}(\mathsf{q}(x)))]_\mathbb{N} \tag{B}$$
$$[\mathsf{a}(\mathsf{q}(x), \mathsf{f}(x))]_\mathbb{N} = 2x^2 + 3x + 2 > 2x^2 + x = [\mathsf{q}(\mathsf{s}(x))]_\mathbb{N} \tag{C}$$
$$[\mathsf{f}(x)]_\mathbb{N} = 2x + 2 > 2x = [\mathsf{a}(x,x)]_\mathbb{N} \tag{D}$$
$$[\mathsf{f}(x)]_\mathbb{N} = 2x + 2 > x = [\mathsf{X}_i(x)]_\mathbb{N} \tag{I$_i$}$$

All other rules of $\mathcal{C}_P$ are weakly oriented. Note that $[t_M]_\mathbb{N} = 0$ for all monomials $M$ with positive coefficient. Hence the rules (J) and (K) are turned into the identity constraint $0 = 0$. In the second step we use the interpretation

$$\begin{array}{llll} \mathsf{0}_\mathbb{N} = 0 & \mathsf{s}_\mathbb{N}(x) = 2x & \mathsf{a}_\mathbb{N}(x,y) = x + y & \mathsf{q}_\mathbb{N}(x) = x \\ \mathsf{A}_\mathbb{N}(x) = x + 1 & \mathsf{B}_\mathbb{N}(x) = x + 1 & \mathsf{f}_\mathbb{N}(x) = x^2 & \mathsf{X}_{i\mathbb{N}}(x) = x \quad \text{for } 1 \leqslant i \leqslant n \end{array}$$

which orients (L) and (M) strictly:

$$[\mathsf{s}(\mathsf{A}(x))]_\mathbb{N} = 2x + 2 > x + 1 = [\mathsf{B}(x)]_\mathbb{N} \tag{L}$$
$$[\mathsf{s}(\mathsf{B}(x))]_\mathbb{N} = 2x + 2 > x + 1 = [\mathsf{A}(x)]_\mathbb{N} \tag{M}$$

In the third step we change the interpretation of $\mathsf{B}$:

$$\begin{array}{llll} \mathsf{0}_\mathbb{N} = 0 & \mathsf{s}_\mathbb{N}(x) = 2x & \mathsf{a}_\mathbb{N}(x,y) = x + y & \mathsf{q}_\mathbb{N}(x) = x \\ \mathsf{A}_\mathbb{N}(x) = x + 1 & \mathsf{B}_\mathbb{N}(x) = x & \mathsf{f}_\mathbb{N}(x) = x^2 & \mathsf{X}_{i\mathbb{N}}(x) = x \quad \text{for } 1 \leqslant i \leqslant n \end{array}$$

This allows to orient (L) and (M) strictly:

$$[\mathsf{A}(\mathsf{s}(t_{P_+}))]_\mathbb{N} = 1 > 0 = [\mathsf{B}(t_{P_-})]_\mathbb{N} \tag{J}$$
$$[\mathsf{A}(\mathsf{s}(t_{P_-}))]_\mathbb{N} = 1 > 0 = [\mathsf{B}(t_{P_+})]_\mathbb{N} \tag{K}$$

The remaining rules (A), (E), (F), (G) and (H$_i$) are strictly oriented using the final interpretation:

$$\begin{array}{lll} \mathsf{0}_\mathbb{N} = 0 & \mathsf{a}_\mathbb{N}(x,y) = x + y & \mathsf{q}_\mathbb{N}(x) = x^2 \\ \mathsf{s}_\mathbb{N}(x) = x + 1 & \mathsf{f}_\mathbb{N}(x) = 3x & \mathsf{X}_{i\mathbb{N}}(x) = x \quad \text{for } 1 \leqslant i \leqslant n \end{array}$$ ◀

▶ **Corollary 22.** *Polynomial termination over* $\mathbb{N}$ *is undecidable for incremental polynomially terminating* TRS*s.*

## 6 Incremental Polynomial Termination is Undecidable

The final result is the undecidability of incremental polynomial termination over $\mathbb{N}$. This time we model the indeterminates in the given polynomial as the degree of the interpretation of the associated function symbols. Due to monotonicity of the interpretations we cannot use polynomials of degree zero. We therefore limit the arguments of the polynomial $P$ to $\mathbb{N}_+$ and use a reduction to (3) from Theorem 4. The idea behind modeling polynomials as degrees of interpretations is illustrated in the following example.

**Table 3** The TRS $\mathcal{D}$.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{q}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{f}(\mathsf{q}(x)))$ | (A) | $\mathsf{f}(x) \to \mathsf{a}(x, x)$ | (C) | $\mathsf{f}(\mathsf{q}(x)) \to \mathsf{m}(x, x)$ | (F) |
| $\mathsf{a}(\mathsf{q}(x), \mathsf{f}(\mathsf{f}(x))) \to \mathsf{q}(\mathsf{a}(x, \mathsf{f}(\mathsf{0})))$ | (B) | $\mathsf{f}(x) \to \mathsf{m}(\mathsf{0}, x)$ | (D) | $\mathsf{m}(x, x) \to \mathsf{q}(x)$ | (G) |
| | | $\mathsf{f}(x) \to \mathsf{m}(x, \mathsf{0})$ | (E) | | |

▶ **Example 23.** Consider the symbols $\mathsf{m}$ and $\mathsf{f}$ where the interpretation of $\mathsf{m}$ is fixed as $\mathsf{m}_{\mathbb{N}}(x, y) = xy + x + y$ (like in Section 4) and $\mathsf{f}_{\mathbb{N}}(x) = dx + c$ is some linear polynomial with coefficients $d, c > 0$. To model that $P(x, y) = 2xy - x \geqslant 0$ for some $x, y \in \mathbb{N}_+$ we also add function symbols for each indeterminate. In this example $\mathsf{X}$ and $\mathsf{Y}$. To model the (positive) coefficients of $P$ we use the variable $x$ for 1 together with the symbol $\mathsf{m}$, which models addition on the level of the degrees of polynomials. Multiplication is modeled as function composition. The rule

$$\mathsf{f}(\mathsf{Y}(\mathsf{X}(\mathsf{m}(x, x)))) \to \mathsf{X}(x)$$

can be oriented only if

$$\deg([\mathsf{f}(\mathsf{Y}(\mathsf{X}(\mathsf{m}(x, x))))]_{\mathbb{N}}) = 2\deg(\mathsf{X}_{\mathbb{N}})\deg(\mathsf{Y}_{\mathbb{N}}) \geqslant \deg(\mathsf{X}_{\mathbb{N}}) = \deg([\mathsf{X}(x)]_{\mathbb{N}})$$

for some polynomials $\mathsf{X}_{\mathbb{N}}$ and $\mathsf{Y}_{\mathbb{N}}$ where $\deg(\mathsf{X}_{\mathbb{N}}), \deg(\mathsf{Y}_{\mathbb{N}}) > 0$. Since otherwise, there will always be some $x \in \mathbb{N}_+$ such that $[\mathsf{f}(\mathsf{Y}(\mathsf{X}(\mathsf{m}(x, x))))]_{\mathbb{N}} < [\mathsf{X}(x)]_{\mathbb{N}}$. Moreover the outermost $\mathsf{f}$ allows us to always chose a large enough $d$ and $c$, such that the rule can be oriented if $2\deg(\mathsf{X}_{\mathbb{N}})\deg(\mathsf{Y}_{\mathbb{N}}) \geqslant \deg \mathsf{X}_{\mathbb{N}}$ independent of the exact shape of $\mathsf{X}_{\mathbb{N}}$ and $\mathsf{Y}_{\mathbb{N}}$. Orienting this rules is therefore possibly if and only if $P(x, y) \geqslant 0$ for some $x, y \in \mathbb{N}_+$.

To constrain the interpretations of the function symbols for this setup to work, we use the following TRS.

▶ **Definition 24.** *Given a polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$, the TRS $\mathcal{D}_P$ is defined over the signature consisting of a constant $\mathsf{0}$, unary function symbols $\mathsf{f}$, $\mathsf{q}$, $\mathsf{X}_1, \ldots, \mathsf{X}_n$, and binary function symbols $\mathsf{a}$ and $\mathsf{m}$. It contains the rewrite rules presented in Table 3, which we denote by $\mathcal{D}$, as well as the single ground rule*

$$\mathsf{f}(\mathsf{0}) \to \mathsf{m}(\mathsf{q}(\mathsf{0}), \mathsf{a}(\mathsf{0}, \mathsf{a}(\mathsf{X}_1(\mathsf{0}), \ldots, \mathsf{a}(\mathsf{X}_{n-1}(\mathsf{0}), \mathsf{X}_n(\mathsf{0})) \ldots))) \qquad \text{(X)}$$

Note that all function symbols with the exception of $\mathsf{f}$ appear in the right-hand side of (X). The importance of this observation we will see later.

▶ **Definition 25.** *Given a monomial $M \in \mathbb{Z}[x_1, \ldots, x_n]$ with positive coefficient, we define the term $t_M$ inductively as follows:*

$$t_M = \begin{cases} x & \text{if } M = 1 \\ \mathsf{m}(x, t_{c-1}) & \text{if } M = c > 1 \\ \mathsf{X}^i(t_{M'}) & \text{if } M = M'x^i \end{cases}$$

*Given a polynomial $P = M_1 + \cdots + M_l \in \mathbb{Z}[x_1, \ldots, x_n]$ with $l \geqslant 1$ and positive coefficients, we define the term $t_P$ inductively as follows:*

$$t_P = \begin{cases} t_{M_1} & \text{if } l = 1 \\ \mathsf{m}(t_{M_1}, t_{P-M_1}) & \text{otherwise} \end{cases}$$

Note that $\mathcal{V}\mathsf{ar}(t_P) = \{x\}$ for any $P \in \mathbb{Z}[x_1,\ldots,x_n]$. So $[t_P]_\mathbb{N}$ is a univariate polynomial.

▶ **Example 26.** For the monomial $M = 3xy^2$ and polynomial $P = 3x^2 + y + 2$ we have

$$t_M = \mathsf{Y}(\mathsf{Y}(\mathsf{X}(\mathsf{m}(x, \mathsf{m}(x,x)))))$$
$$t_P = \mathsf{m}(\mathsf{X}(\mathsf{X}(\mathsf{m}(x,\mathsf{m}(x,x)))), \mathsf{m}(\mathsf{Y}(x), \mathsf{m}(x,x)))$$

▶ **Lemma 27.** *If $\mathsf{m}_\mathbb{N}(x,y) = m_3xy + m_2x + m_1y + m_0$ with $m_3 > 0$ then*

$$\deg([t_M]_\mathbb{N}) = c \cdot \deg(\mathsf{X}_{1\mathbb{N}})^{i_1} \cdots \deg(\mathsf{X}_{n\mathbb{N}})^{i_n} = M(\deg(\mathsf{X}_{1\mathbb{N}}),\ldots,\deg(\mathsf{X}_{n\mathbb{N}}))$$

*for monomials $M = c\,x_1^{i_1}\cdots x_n^{i_n}$ with $c > 0$, and*

$$\deg([t_P]_\mathbb{N}) = \deg([t_{M_1}]_\mathbb{N}) + \cdots + \deg([t_{M_k}]_\mathbb{N}) = P(\deg(\mathsf{X}_{1\mathbb{N}}),\ldots,\deg(\mathsf{X}_{n\mathbb{N}}))$$

*for polynomials $P = M_1 + \cdots + M_l$ with $l \geqslant 1$ and positive coefficients.*

**Proof.** We prove the statement for monomials $M$. If $M = 1$ then $t_M = x$ and $[t_M]_\mathbb{N} = x$ and thus $\deg([t_M]_\mathbb{N}) = 1$. If $M = c > 1$ then $t_M = \mathsf{m}(x, t_{c-1})$ and, due to the assumption concerning the interpretation of $\mathsf{m}$, $[t_M]_\mathbb{N} = m_3x[t_{c-1}]_\mathbb{N} + m_2x + m_1[t_{c-1}]_\mathbb{N} + m_0$ with $m_3 > 0$. We obtain $\deg([t_{c-1}]_\mathbb{N}) = c - 1$ from the induction hypothesis. Hence

$$\deg([t_M]_\mathbb{N}) = 1 + (c-1) = c = M(\deg(\mathsf{X}_{1\mathbb{N}}),\ldots,\deg(\mathsf{X}_{n\mathbb{N}}))$$

If $M = M'x^i$ then $t_M = \mathsf{X}^i(t_{M'})$. Without loss of generality we assume that $x = x_n$ and $M' \in \mathbb{Z}[x_1,\ldots,x_{n-1}]$. We have $[t_M]_\mathbb{N} = (\mathsf{X}_{n\mathbb{N}})^i[t_{M'}]_\mathbb{N}$. The induction hypothesis yields

$$\deg([t_{M'}]_\mathbb{N}) = c \cdot \deg(\mathsf{X}_{1\mathbb{N}})^{i_1} \cdots \deg(\mathsf{X}_{n-1\mathbb{N}})^{i_{n-1}} = M'(\deg(\mathsf{X}_{1\mathbb{N}}),\ldots,\deg(\mathsf{X}_{n-1\mathbb{N}}))$$

and thus $\deg([t_M]_\mathbb{N}) = c \cdot \deg(\mathsf{X}_{1\mathbb{N}})^{i_1} \cdots \deg(\mathsf{X}_{n\mathbb{N}})^{i_n} = M(\deg(\mathsf{X}_{1\mathbb{N}}),\ldots,\deg(\mathsf{X}_{n\mathbb{N}}))$ by setting $i_n = i$. The statement for polynomials is an easy consequence of the one for monomials. ◀

▶ **Example 28.** Consider the polynomial $P = 2x^2 + y + 1$ and suppose $\mathsf{m}_\mathbb{N}(x,y) = xy + x + y$, $\mathsf{X}_\mathbb{N}(x) = x^2$ and $\mathsf{Y}_\mathbb{N}(x) = x^3$. We have

$$t_P = \mathsf{m}(\mathsf{X}(\mathsf{X}(\mathsf{m}(x,x))), \mathsf{m}(\mathsf{Y}(x), x))$$
$$[t_P]_\mathbb{N} = ((x^2 + 2x)^2)^2(x^3x + x^3 + x) + ((x^2 + 2x)^2)^2) + (x^3x + x^3 + x)$$

Note that $\deg([t_P]_\mathbb{N}) = 12 = P(2,3)$. If we change the interpretations of $\mathsf{X}$ and $\mathsf{Y}$ to $\mathsf{X}_\mathbb{N}(x) = x^5$ and $\mathsf{Y}_\mathbb{N}(x) = x^4$ then

$$[t_P]_\mathbb{N} = ((x^2 + 2x)^5)^5(x^4x + x^4 + x) + ((x^2 + 2x)^5)^5) + (x^4x + x^4 + x)$$

and $\deg([t_P]_\mathbb{N}) = 55 = P(5,4)$.

▶ **Definition 29.** *The* TRS $\mathcal{D}_P$ *is the extension of* $\mathcal{D} \cup \{(\mathrm{X})\}$ *with the single rule*

$$\mathsf{f}(t_{P_+}) \to t_{P_-} \tag{H}$$

Since $t_0$ is undefined in Definition 25, the TRS $\mathcal{D}_P$ is defined only when $P$ contains both monomials with positive and with negative coefficients. Since Hilbert's 10th problem is trivially decidable for polynomials with only positive (negative) coefficients, this entails no loss of generality.

▶ **Example 30.** The TRS $\mathcal{D}_{x^2-2xy+3}$ consists of the rules in Table 3 extended with

$$\mathsf{f}(\mathsf{0}) \to \mathsf{m}(\mathsf{q}(\mathsf{0}), \mathsf{a}(\mathsf{0}, \mathsf{a}(\mathsf{X}(\mathsf{0}), \mathsf{Y}(\mathsf{0})))) \qquad \mathsf{f}(\mathsf{m}(\mathsf{X}(\mathsf{X}(x)), \mathsf{m}(x, \mathsf{m}(x, x)))) \to \mathsf{Y}(\mathsf{X}(\mathsf{m}(x, x)))$$

The following lemma is used in the proof of the main result of this section.

▶ **Lemma 31.** *If* $\deg(P) \geqslant \deg(Q)$ *for univariate polynomials* $P, Q \in \mathbb{Z}[x]$ *with positive coefficients then*

$$c \cdot P(x) + c > Q(x)$$

*for some* $c \in \mathbb{N}$ *and all* $x \in \mathbb{N}$.

**Proof.** Let $k$ be the degree of $Q$. So $Q(x) = a_k x^k + \cdots + a_1 x^1 + a_0$ for some coefficients $a_0, \ldots, a_k \in \mathbb{N}$ with $a_k \neq 0$. Define $c = a_k + \cdots + a_1 + a_0 + 1$. We have

$$c \cdot P(x) + c \geqslant cx^k + c \geqslant Q(x) + 1 > Q(x)$$

for all $x \in \mathbb{N}$. ◀

▶ **Theorem 32.** *For any polynomial* $P \in \mathbb{Z}[x_1, \ldots, x_n]$ *with both positive and negative coefficients, the* TRS $\mathcal{D}_P$ *is incremental polynomially terminating over* $\mathbb{N}$ *if and only if* $P(a_1, \ldots, a_n) \geqslant 0$ *for some* $a_1, \ldots, a_n \in \mathbb{N}_+$.

**Proof.** For the only-if direction, suppose $\mathcal{D}_P$ is incremental polynomially terminating over $\mathbb{N}$. From (A) we infer $\deg(\mathsf{f}_\mathbb{N}) = 1$. So $\mathsf{f}_\mathbb{N}(x) = f_1 x + f_0$ with $f_1 > 0$. From rule (C) we obtain $\deg(\mathsf{a}_\mathbb{N}) = 1$ and thus $\mathsf{a}_\mathbb{N}(x, y) = a_2 x + a_1 y + a_0$ with $a_2, a_1 > 0$ and subsequently $f_1 > (a_2 + a_1) \geqslant 2$. Because $\mathcal{D}_P$ is incremental polynomially terminating, at least one of its rewrite rules is oriented strictly. This is possible only if the constant part of some interpretation function is positive. Now consider rule (X). Since it contains all function symbols, either $f_0 > 0$ or

$$[\mathsf{m}(\mathsf{q}(\mathsf{0}), \mathsf{a}(\mathsf{0}, \mathsf{a}(\mathsf{X}_1(\mathsf{0}), \ldots, \mathsf{a}(\mathsf{X}_{n-1}(\mathsf{0}), \mathsf{X}_n(\mathsf{0})) \ldots )))]_\mathbb{N} > 0$$

In both cases we obtain $[\mathsf{f}(\mathsf{0})]_\mathbb{N} > 0$. Consider rule (A) again. If $\mathsf{q}_\mathbb{N}(x)$ is linear then $f_1 = 1$, contradicting $f_1 \geqslant 2$. So $\deg(\mathsf{q}_\mathbb{N}) \geqslant 2$. From (B) we infer

$$a_1 \mathsf{f}_\mathbb{N}(\mathsf{f}_\mathbb{N}(x)) + a_0 \geqslant \mathsf{q}_\mathbb{N}(a_2 x + a_1 [\mathsf{f}(\mathsf{0})]_\mathbb{N} + a_0) - a_2 \mathsf{q}_\mathbb{N}(x)$$

Abbreviating $a_1 [\mathsf{f}(\mathsf{0})]_\mathbb{N} + a_0$ to $d$ and letting $\mathsf{q}_\mathbb{N}(x) = q_k x^k + \cdots + q_1 x^1 + q_0$ with $k \geqslant 2$, the expression $\mathsf{q}_\mathbb{N}(a_2 x + d) - a_2 \mathsf{q}_\mathbb{N}(x)$ evaluates to

$$\begin{aligned}
\sum_{i=0}^{k} q_i (a_2 x + d)^i - a_2 \sum_{i=0}^{k} q_i x^i &= \sum_{i=0}^{k} q_i \sum_{j=0}^{i} \binom{i}{j} a_2^j x^j d^{i-j} - a_2 \sum_{i=0}^{k} q_i x^i \\
&= \sum_{i=0}^{k} q_i \left( a_2^i x^i + \sum_{j=0}^{i-1} \binom{i}{j} a_2^j x^j d^{i-j} \right) - a_2 \sum_{i=0}^{k} q_i x^i \\
&= \sum_{i=0}^{k} q_i \sum_{j=0}^{i-1} \binom{i}{j} a_2^j x^j d^{i-j} - \sum_{i=0}^{k} q_i (a_2^i - a_2) x^i
\end{aligned}$$

Note that $d > 0$ because $[\mathsf{f}(\mathsf{0})]_\mathbb{N} > 0$ and $a_2 > 0$. The degree of the left sum is $k - 1$ whereas the right sum has degree 0 if $a_2 = 1$ and $k$ if $a_2 \neq 1$. Since the degree is bounded by $\deg(\mathsf{f}_\mathbb{N})^2 = 1$, we must have $a_2 = 1$. Hence

$$1 \geqslant \deg(\mathsf{q}_\mathbb{N}(a_2 x + a_1 [\mathsf{f}(\mathsf{0})]_\mathbb{N} + a_0) - a_2 \mathsf{q}_\mathbb{N}(x)) = \deg(\mathsf{q}_\mathbb{N}) - 1$$

and thus $\deg(\mathsf{q}_{\mathbb{N}}) = 2$. Rules (F) and (G) ensure $\deg(\mathsf{m}_{\mathbb{N}}(x,x)) = 2$ and thus we may write $\mathsf{m}_{\mathbb{N}}(x,y) = m_5x^2 + m_4y^2 + m_3xy + m_2x + m_1y + m_0$. Considering rule (D) yields

$$1 \geqslant \deg(\mathsf{m}_{\mathbb{N}}(\mathsf{0}_{\mathbb{N}},x)) = \deg(m_5\mathsf{0}_{\mathbb{N}}^2 + m_4x^2 + (m_3\mathsf{0}_{\mathbb{N}} + m_1)x + m_2\mathsf{0}_{\mathbb{N}} + m_0)$$

and thus $m_4 = 0$. Similarly, rule (E) yields $m_5 = 0$. Hence $m_3 > 0$ for otherwise $\deg(\mathsf{m}_{\mathbb{N}}(x,x)) = 2$ does not hold. From rule (H) with $\deg(\mathsf{f}_{\mathbb{N}}) = 1$ we obtain $\deg([t_{P_+}]_{\mathbb{N}}) \geqslant \deg([t_{P_-}]_{\mathbb{N}})$. Subsequently applying Lemma 27 to the terms $t_{P_+}$ and $t_{P_-}$ results in

$$P_+(\deg(\mathsf{X}_{1\mathbb{N}}),\dots,\deg(\mathsf{X}_{n\mathbb{N}})) \geqslant P_-(\deg(\mathsf{X}_{1\mathbb{N}}),\dots,\deg(\mathsf{X}_{n\mathbb{N}}))$$

Hence $P(\deg(\mathsf{X}_{1\mathbb{N}}),\dots,\deg(\mathsf{X}_{n\mathbb{N}})) \geqslant 0$ as desired.

For the if direction, suppose $P(a_1,\dots,a_n) \geqslant 0$ for some $a_1,\dots,a_n \in \mathbb{N}_+$. The interpretation

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{a}_{\mathbb{N}}(x,y) = x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = x^2 + x$$
$$\mathsf{f}_{\mathbb{N}}(x) = fx + f \qquad \mathsf{m}_{\mathbb{N}}(x,y) = xy + x + y \qquad \mathsf{X}_{i\mathbb{N}}(x) = x^{a_i} \quad \text{for } 1 \leqslant i \leqslant n$$

with $f \geqslant 2$ orients the rules of $\mathcal{D} \cup \{(\mathrm{X})\}$ as follows:

$$\begin{aligned}
\mathsf{q}(\mathsf{f}(x)) &\to \mathsf{f}(\mathsf{f}(\mathsf{q}(x))) & (fx+f)^2 + fx + f &\geqslant f^2(x^2+x) + f^2 + f & \text{(A)}\\
\mathsf{a}(\mathsf{q}(x),\mathsf{f}(\mathsf{f}(x))) &\to \mathsf{q}(\mathsf{a}(x,\mathsf{f}(\mathsf{0}))) & x^2 + x + f^2x + f^2 + f &\geqslant (x+f)^2 + x + f & \text{(B)}\\
\mathsf{f}(x) &\to \mathsf{a}(x,x) & fx + f &> 2x & \text{(C)}\\
\mathsf{f}(x) &\to \mathsf{m}(\mathsf{0},x) & fx + f &> x & \text{(D)}\\
\mathsf{f}(x) &\to \mathsf{m}(x,\mathsf{0}) & fx + f &> x & \text{(E)}\\
\mathsf{f}(\mathsf{q}(x)) &\to \mathsf{m}(x,x) & f(x^2+x) + f &> x^2 + 2x & \text{(F)}\\
\mathsf{m}(x,x) &\to \mathsf{q}(x) & x^2 + 2x &\geqslant x^2 + x & \text{(G)}\\
\mathsf{f}(\mathsf{0}) &\to \mathsf{m}(\mathsf{q}(\mathsf{0}),\mathsf{a}(\mathsf{0},\mathsf{a}(\mathsf{X}_1(\mathsf{0}),\dots,\mathsf{a}(\mathsf{X}_{n-1}(\mathsf{0}),\mathsf{X}_n(\mathsf{0}))\dots))) & f &> 0 & \text{(X)}
\end{aligned}$$

The assumption $P(a_1,\dots,a_n) \geqslant 0$ in connection with Lemma 27 yields

$$0 \leqslant P_+(a_1,\dots,a_n) - P_-(a_1,\dots,a_n) = \deg([t_{P_+}]_{\mathbb{N}}) - \deg([t_{P_-}]_{\mathbb{N}})$$

Since $[t_{P_+}]_{\mathbb{N}}$ and $[t_{P_-}]_{\mathbb{N}}$ are univariate polynomials, we can apply Lemma 31. This yields a $c \in \mathbb{N}$ such that $c[t_{P_+}]_{\mathbb{N}} + c > [t_{P_-}]_{\mathbb{N}}$. Hence, by choosing $f = \max(c,2)$, the rule (H) is strictly oriented. This concludes the first step in the incremental polynomial termination proof of $\mathcal{D}_P$. The interpretation

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{a}_{\mathbb{N}}(x,y) = 2x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = 3x + 2$$
$$\mathsf{f}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{m}_{\mathbb{N}}(x,y) = 2x + y + 3$$

orients the remaining rules (A), (B) and (G):

$$\begin{aligned}
\mathsf{q}(\mathsf{f}(x)) &\to \mathsf{f}(\mathsf{f}(\mathsf{q}(x))) & 3x + 5 &> 3x + 4 & \text{(A)}\\
\mathsf{a}(\mathsf{q}(x),\mathsf{f}(\mathsf{f}(x))) &\to \mathsf{q}(\mathsf{a}(x,\mathsf{f}(\mathsf{0}))) & 7x + 6 &> 6x + 5 & \text{(B)}\\
\mathsf{m}(x,x) &\to \mathsf{q}(x) & 3x + 3 &> 3x + 2 & \text{(G)}
\end{aligned}$$

Hence $\mathcal{D}_P$ is incremental polynomially terminating over $\mathbb{N}$. ◀

▶ **Corollary 33.** *Incremental polynomial termination over $\mathbb{N}$ is an undecidable property of finite* TRS*s.*

We do not know whether the TRS $\mathcal{D}_P$ is terminating (independent of $P$). Hence $\mathcal{D}_P$ cannot be used to strengthen the result of Corollary 33 to terminating TRSs. However, a small modification is sufficient to obtain this result.

▶ **Definition 34.** *The* TRS $\mathcal{D}'_P$ *is defined as* $\{\mathsf{D}^\ell_r(\ell) \to \mathsf{D}^\ell_r(r) \mid \ell \to r \in \mathcal{D}_P\}$.

So each rule $\ell \to r$ of $\mathcal{D}_P$ is placed under a designated unary function symbol $\mathsf{D}^\ell_r$. The indices $\ell$ and $r$ ensure that different rules are rooted by different symbols. The proof of Theorem 32 is not affected by this modification, since monotonicity implies that $\mathsf{D}^\ell_r(x) > \mathsf{D}^\ell_r(y)$ if and only if $x > y$.

▶ **Lemma 35.** *The* TRS $\mathcal{D}'_P$ *is terminating.*

**Proof.** Let the signature of $\mathcal{D}'_P$ be denoted by $\mathcal{F}$. We prove termination using a well-founded monotone $\mathcal{F}$-algebra $\mathcal{A} = (\mathbb{N} \times \mathcal{F}, >_\mathcal{A})$, where $(n, f) >_\mathcal{A} (m, g)$ if $f = g$ and $n >_\mathbb{N} m$. The interpretation functions for the symbols in $\mathcal{D}_P$ are (with $1 \leqslant i \leqslant n$)

$$\mathsf{0}_\mathcal{A} = (1, 0) \qquad \mathsf{a}_\mathcal{A}((x, f), (y, g)) = (x + y + 1, \mathsf{a}) \qquad \mathsf{q}_\mathcal{A}((x, f)) = (x + 1, \mathsf{q})$$
$$\mathsf{f}_\mathcal{A}((x, f)) = (x + 1, \mathsf{f}) \quad \mathsf{m}_\mathcal{A}((x, f), (y, g)) = (x + y + 1, \mathsf{m}) \quad \mathsf{X}_{i\mathcal{A}}((x, f)) = (x + 1, \mathsf{X}_i)$$

Intuitively these interpretations keep track of the size and the root symbol of the term. The interpretations of the symbols $\mathsf{D}^\ell_r$ ensure that they weigh more when appearing on the left and are defined as

$$\mathsf{D}^\ell_{r\mathcal{A}}((x, f)) = \begin{cases} (|r| \cdot x + 1, \mathsf{D}^\ell_r) & \text{if } f = \mathsf{root}(\ell) \\ (x + 1, \mathsf{D}^\ell_r) & \text{otherwise} \end{cases}$$

where $|r|$ denotes the size of the term $r$. One easily verifies that all rewrite rules in $\mathcal{D}'_P$ are oriented strictly with respect to $>_\mathcal{A}$; note that for right-hand sides $\mathsf{D}^\ell_r(r)$ the second case in the interpretation of $\mathsf{D}^\ell_r$ applies, except when $r = t_1 = x$ in which case the rule is oriented based on the size of the terms alone. ◀

The second component in the interpretations in the above proof simulates root-labeling [16] and the lemma can also be shown using semantic root-labeling in connection with LPO [17, 13].

▶ **Corollary 36.** *Incremental polynomial termination over* ℕ *is an undecidable property of terminating* TRS*s.*

## 7 Conclusion

In this paper we proved the undecidability of polynomial termination over ℕ for TRSs that are incremental polynomially terminating over ℕ. We also proved that incremental polynomial termination over ℕ is an undecidable property of terminating TRSs. The proofs remain valid if we restrict to polynomial interpretations with natural numbers as coefficients. A simple tool that generates the TRSs $\mathcal{R}_P$, $\mathcal{C}_P$ and $\mathcal{D}_P$ given a polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$ is available[1] and useful for tool builders and competition organizers.

As possible future work regarding decidability we mention weakly monotone interpretations over ℕ as used in a dependency pairs setting [1]. Polynomial interpretations over ℚ and ℝ ([11, 15]) are also of interest. Moreover, matrix [5] and arctic [9] interpretations are under-explored as far as decidability issues are concerned.

[1] `https://github.com/fabeulous/pt-hilbert-encodings`

## References


1. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.
3. Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987. doi:10.1016/0167-6423(87)90030-X.
4. Martin Davis. Hilbert's tenth problem is unsolvable. *The American Mathematical Monthly*, 80(3):233–269, 1973. doi:10.1080/00029890.1973.11993265.
5. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of rewrite systems. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008. doi:10.1007/s10817-007-9087-9.
6. Alfons Geser. *Relative Termination.* PhD thesis, University of Passau, Germany, 1990. doi:10.18725/OPARU-2427.
7. David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8(10):437–479, 1902. doi:10.1090/S0002-9904-1902-00923-3.
8. Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations (preliminary version). In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177, 1989. doi:10.1007/3-540-51081-8_107.
9. Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.
10. Dallas Lankford. On proving term rewrite systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
11. Salvador Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 17(1):49–73, 2006. doi:10.1007/s00200-005-0189-5.
12. Yuri Y. Matijasevic. Enumerable sets are diophantine (translated from Russian). In *Soviet Mathematics Doklady*, volume 11, pages 354–358, 1970.
13. Aart Middeldorp, Hitoshi Ohsaki, and Hans Zantema. Transforming termination by self-labelling. In *Proceedings of the 13th International Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 373–387, 1996. doi:10.1007/3-540-61511-3_101.
14. Fabian Mitterwallner and Aart Middeldorp. Polynomial termination over N is undecidable. In Samir Genaim, editor, *Proceedings of the 17th International Workshop on Termination*, pages 21–26, 2021.
15. Friedrich Neurauter and Aart Middeldorp. Polynomial interpretations over the natural, rational and real numbers revisited. *Logical Methods in Computer Science*, 10(3:22):1–28, 2014. doi:10.2168/LMCS-10(3:22)2014.
16. Christian Sternagel and Aart Middeldorp. Root-labeling. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350, 2008. doi:10.1007/978-3-540-70590-1_23.
17. Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995. doi:10.3233/FI-1995-24124.
18. Hans Zantema. Termination. In *Term Rewriting Systems*, chapter 6, pages 181–259. Cambridge University Press, 2003.

# Compositional Confluence Criteria

**Kiraku Shintani**
Japan Advanced Institute of Science and Technology, Ishikawa, Japan

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

**Abstract**

We show how confluence criteria based on decreasing diagrams are generalized to ones composable with other criteria. For demonstration of the method, the confluence criteria of orthogonality, rule labeling, and critical pair systems for term rewriting are recast into composable forms. In addition to them, we prove that Toyama's parallel closedness result based on parallel critical pairs subsumes his almost parallel closedness theorem.

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

**Keywords and phrases** term rewriting, confluence, decreasing diagrams

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.28

**Funding** *Nao Hirokawa*: JSPS KAKENHI Grant Numbers 22K11900.

**Acknowledgements** We are grateful to Jean-Pierre Jouannaud, Vincent van Oostrom, and Yoshihito Toyama for their valuable comments on preliminary results of this work. We thank the reviewers for their thorough reading and suggestions, which helped us to improve the presentation.

## 1 Introduction

Confluence is a property of rewriting that ensures uniqueness of computation results. In the last decades, various proof methods for confluence of term rewrite systems have been developed. They are roughly classified to three groups: (direct) confluence criteria based on critical pair analysis [18, 15, 30, 32, 10, 36, 23, 37, 40], decomposition methods based on modularity and commutation [31, 3, 27], and transformation methods based on simulation of rewriting [2, 17, 20, 27].

In this paper we present a confluence analysis based on *compositional* confluence criteria. Here a compositional criterion means a sufficient condition that, given a rewrite system $\mathcal{R}$ and its subsystem $\mathcal{C} \subseteq \mathcal{R}$, the confluence of $\mathcal{C}$ implies that of $\mathcal{R}$. Since such a subsystem can be analyzed by any other (compositional) confluence criterion, compositional criteria can be seen as a combination method for confluence analysis. Because the empty system is confluent, by taking the empty subsystem $\mathcal{C}$ compositional criteria can be used as ordinary (direct) confluence criteria.

In order to develop compositional confluence criteria we revisit van Oostrom's decreasing diagram technique [35, 37], which is known as a powerful confluence criterion for abstract rewrite systems. Most of existing confluence criteria for left-linear rewrite systems, including the ones listed above, can be proved by decreasingness of parallel steps or multi-steps. Recasting the decreasing diagram technique as a compositional criterion, we demonstrate how confluence criteria based on decreasing diagrams can be reformulated as compositional versions. We pick up the confluence criteria by orthogonality [26], rule labeling [40], and critical pair systems [11].

© Kiraku Shintani and Nao Hirokawa;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 28; pp. 28:1–28:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In addition, we elucidate the hierarchy of Toyama's two parallel closedness theorems [30, 32] and rule labeling based on parallel critical pairs [40]. As a consequence, it turns out that rule labeling and its compositional version are generalizations of Huet's and Toyama's (almost) parallel closedness theorems.

The remaining part of the paper is organized as follows: In Section 2 we recall notions from rewriting. In Section 3 we show that Toyama's almost parallel closedness is subsumed by his earlier result based on parallel critical pairs. In Section 4, we introduce an abstract criterion for our approach, and in the subsequent three sections we derive compositional criteria from the confluence criteria of orthogonality (Section 5), rule labeling (Section 6), and the criterion by critical pair systems (Section 7). Section 8 reports experimental results. Discussing related work and potential future work in Section 9, we conclude the paper.

## 2 Preliminaries

Throughout the paper, we assume familiarity with abstract rewriting and term rewriting [4, 29]. We just recall some basic notions and notations for rewriting and confluence.

An ($I$-indexed) *abstract rewrite system* (ARS) $\mathcal{A}$ is a pair $(A, \{\to_\alpha\}_{\alpha \in I})$ consisting of a set $A$ and the family of relations $\to_\alpha$ on $A$. Given a subset $J$ of $I$, we write $x \to_J y$ if $x \to_\alpha y$ for some index $\alpha \in J$. The relation $\to_I$ is referred to as $\to_\mathcal{A}$. An ARS $\mathcal{A}$ is called *confluent* or *locally confluent* if ${}^*_\mathcal{A}\!\leftarrow \cdot \to^*_\mathcal{A} \subseteq \to^*_\mathcal{A} \cdot {}^*_\mathcal{A}\!\leftarrow$ or ${}_\mathcal{A}\!\leftarrow \cdot \to_\mathcal{A} \subseteq \to^*_\mathcal{A} \cdot {}^*_\mathcal{A}\!\leftarrow$ holds, respectively. We say that ARSs $\mathcal{A}$ and $\mathcal{B}$ *commute* if ${}^*_\mathcal{A}\!\leftarrow \cdot \to^*_\mathcal{B} \subseteq \to^*_\mathcal{B} \cdot {}^*_\mathcal{A}\!\leftarrow$ holds. A conversion of form $b \;{}_\mathcal{A}\!\leftarrow a \to_\mathcal{B} c$ is called a *local peak* (or simply a *peak*) between $\mathcal{A}$ and $\mathcal{B}$. An ARS $\mathcal{A}$ is *terminating* if there exists no infinite sequence $a_0 \to_\mathcal{A} a_1 \to_\mathcal{A} \cdots$. We define $\to_{\mathcal{A}/\mathcal{B}}$ as $\to^*_\mathcal{B} \cdot \to_\mathcal{A} \cdot \to^*_\mathcal{B}$. We say that $\mathcal{A}$ is *relatively terminating* with respect to $\mathcal{B}$, or simply $\mathcal{A}/\mathcal{B}$ is *terminating*, if $\to_{\mathcal{A}/\mathcal{B}}$ is terminating.

Positions are sequences of positive integers. The empty sequence $\epsilon$ is called the *root* position. We write $p \cdot q$ or simply $pq$ for the concatenation of positions $p$ and $q$. The prefix order $\leqslant$ on positions is defined as $p \leqslant q$ if $p \cdot p' = q$ for some $p'$. We say that positions $p$ and $q$ are *parallel* if $p \nleqslant q$ and $q \nleqslant p$. A set of positions is called *parallel* if all its elements are so.

Terms are built from a signature $\mathcal{F}$ and a countable set $\mathcal{V}$ of variables satisfying $\mathcal{F} \cap \mathcal{V} = \varnothing$. The set of all terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $t$ be a term. The set of all variables in $t$ is denoted by $\mathcal{V}\mathsf{ar}(t)$, and the set of all function positions and the set of variable positions in $t$ by $\mathcal{P}\mathsf{os}_\mathcal{F}(t)$ and $\mathcal{P}\mathsf{os}_\mathcal{V}(t)$, respectively. The *subterm* of $t$ at position $p$ is denoted by $t|_p$. It is a *proper* subterm if $p \neq \epsilon$. By $t[u]_p$ we denote the term that results from replacing the subterm of $t$ at $p$ by term $u$. The size $|t|$ of $t$ is the number of occurrences of functions symbols and variables in $t$. A term $t$ is said to be *linear* if every variable in $t$ occurs exactly once.

A *substitution* is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ whose domain $\mathcal{D}\mathsf{om}(\sigma)$ is finite. Here $\mathcal{D}\mathsf{om}(\sigma)$ stands for the set $\{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. The term $t\sigma$ is defined as $\sigma(t)$ for $t \in \mathcal{V}$, and $f(t_1\sigma, \ldots, t_n\sigma)$ for $t = f(t_1, \ldots, t_n)$. A term $u$ is called an *instance* of $t$ if $u = t\sigma$ for some $\sigma$. A substitution is called a *renaming* if it is a bijection on variables.

A *term rewrite system* (TRS) over $\mathcal{F}$ is a set of rewrite rules. Here a pair $(\ell, r)$ of terms is a *rewrite rule* or simply a *rule* if $\ell \notin \mathcal{V}$ and $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell)$. We denote it by $\ell \to r$. The rewrite relation $\to_\mathcal{R}$ of a TRS $\mathcal{R}$ is defined on terms as follows: $s \to_\mathcal{R} t$ if $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$ for some rule $\ell \to r \in \mathcal{R}$, position $p$, and substitution $\sigma$. We write $s \xrightarrow{p}_\mathcal{R} t$ if the rewrite position $p$ is relevant. We call subsets of $\mathcal{R}$ *subsystems*. A TRS $\mathcal{R}$ is *left-linear* if $\ell$ is linear for all $\ell \to r \in \mathcal{R}$. Since any TRS $\mathcal{R}$ can be regarded as the ARS $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \{\to_\mathcal{R}\})$, we

use notions and notations of ARSs for TRSs. For instance, a TRS $\mathcal{R}$ is (locally) confluent if the ARS $(\mathcal{T}(\mathcal{F},\mathcal{V}),\{\to_{\mathcal{R}}\})$ is so. Similarly, two TRSs commute if their corresponding ARSs commute.

Local confluence of TRSs is characterized by notion of critical pair. We say that a rule $\ell_1 \to r_1$ is a *variant* of a rule $\ell_2 \to r_2$ if $\ell_1\rho = \ell_2$ and $r_1\rho = r_2$ for some renaming $\rho$.

▶ **Definition 1.** *Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs. Suppose that the following conditions hold:*

- *$\ell_1 \to r_1$ and $\ell_2 \to r_2$ are variants of rules in $\mathcal{R}$ and in $\mathcal{S}$, respectively,*
- *$\ell_1 \to r_1$ and $\ell_2 \to r_2$ have no common variables,*
- *$p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$,*
- *$\sigma$ is a most general unifier of $\ell_1$ and $\ell_2|_p$, and*
- *if $p = \epsilon$ then $\ell_1 \to r_1$ is not a variant of $\ell_2 \to r_2$.*

*The local peak $(\ell_2\sigma)[r_1\sigma]_p \;{}_{\mathcal{R}}\!\overset{p}{\leftarrow} \ell_2\sigma \overset{\epsilon}{\to}_{\mathcal{S}} r_2\sigma$ is called a* critical peak *between $\mathcal{R}$ and $\mathcal{S}$. When $t \;{}_{\mathcal{R}}\!\overset{p}{\leftarrow}\; s \overset{\epsilon}{\to}_{\mathcal{S}} u$ is a critical peak, the pair $(t,u)$ is called a* critical pair*. To clarify the orientation of the pair, we denote it as the binary relation $t \;{}_{\mathcal{R}}\!\overset{p}{\leftarrow}\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{S}} u$, see [6]. Moreover, we write $t \;{}_{\mathcal{R}}\!\leftarrow\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{S}} u$ if $t \;{}_{\mathcal{R}}\!\overset{p}{\leftarrow}\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{S}} u$ for some position $p$.*

▶ **Theorem 2** ([15])**.** *A TRS $\mathcal{R}$ is locally confluent if and only if ${}_{\mathcal{R}}\!\leftarrow\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{R}} \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^{*}\!\leftarrow$ holds.*

Combining it with Newman's Lemma [21], we obtain Knuth and Bendix' criterion [18].

▶ **Theorem 3** ([18])**.** *A terminating TRS $\mathcal{R}$ is confluent if and only if ${}_{\mathcal{R}}\!\leftarrow\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{R}} \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^{*}\!\leftarrow$ holds.*

We define the parallel step relation, which plays a key role in analysis of local peaks.

▶ **Definition 4.** *Let $\mathcal{R}$ be a TRS and let $P$ be a set of parallel positions. The* parallel step *$\overset{P}{\twoheadrightarrow}_{\mathcal{R}}$ is inductively defined on terms as follows:*

- *$x \overset{P}{\twoheadrightarrow}_{\mathcal{R}} x$ if $x$ is a variable and $P = \varnothing$.*
- *$\ell\sigma \overset{P}{\twoheadrightarrow}_{\mathcal{R}} r\sigma$ if $\ell \to r$ is an $\mathcal{R}$-rule, $\sigma$ is a substitution, and $P = \{\epsilon\}$.*
- *$f(s_1,\ldots,s_n) \overset{P}{\twoheadrightarrow}_{\mathcal{R}} f(t_1,\ldots,t_n)$ if $f$ is an $n$-ary function symbol in $\mathcal{F}$, $s_i \overset{P_i}{\twoheadrightarrow}_{\mathcal{R}} t_i$ holds for all $1 \leqslant i \leqslant n$, and $P = \{i \cdot p \mid 1 \leqslant i \leqslant n \text{ and } p \in P_i\}$.*

*We write $s \twoheadrightarrow_{\mathcal{R}} t$ if $s \overset{P}{\twoheadrightarrow}_{\mathcal{R}} t$ for some set $P$ of positions.*

Note that $\twoheadrightarrow_{\mathcal{R}}$ is reflexive and the inclusions $\to_{\mathcal{R}} \subseteq \twoheadrightarrow_{\mathcal{R}} \subseteq \to_{\mathcal{R}}^*$ hold. As the latter entails $\to_{\mathcal{R}}^* = \twoheadrightarrow_{\mathcal{R}}^*$, we obtain the following useful characterizations.

▶ **Lemma 5.** *A TRS $\mathcal{R}$ is confluent if and only if $\twoheadrightarrow_{\mathcal{R}}$ is confluent. Similarly, TRSs $\mathcal{R}$ and $\mathcal{S}$ commute if and only if $\twoheadrightarrow_{\mathcal{R}}$ and $\twoheadrightarrow_{\mathcal{S}}$ commute.*

## 3 Parallel Closedness

Toyama made two variations of Huet's parallel closedness theorem [15] in 1981 [30] and in 1988 [32], but their relation has not been known. In this section we recall his and related results, and then show that Toyama's earlier result subsumes the later one. For brevity we omit the subscript $\mathcal{R}$ from $\to_{\mathcal{R}}$, $\twoheadrightarrow_{\mathcal{R}}$, and ${}_{\mathcal{R}}\!\leftarrow\!\rtimes\!\overset{\epsilon}{\to}_{\mathcal{R}}$ when it is clear from the contexts.

▶ **Definition 6** ([15])**.** *A TRS is* parallel closed *if $\leftarrow\!\rtimes\!\overset{\epsilon}{\to} \subseteq \twoheadrightarrow$ holds.*

▶ **Theorem 7** ([15])**.** *A left-linear TRS is confluent if it is parallel closed.*

In 1988, Toyama showed that the closing form for *overlay* critical pairs, originating from root overlaps, can be relaxed. We write $t \xleftarrow{>\epsilon} \rtimes \xrightarrow{\epsilon} u$ if $t \xleftarrow{p} \rtimes \xrightarrow{\epsilon} u$ holds for some $p > \epsilon$.

▶ **Definition 8** ([32]). *A TRS is* almost parallel closed *if* $\xleftarrow{\epsilon} \rtimes \xrightarrow{\epsilon} \subseteq \mathrel{\parallel\!\!\!\rightarrow} \cdot {}^{*}\!\leftarrow$ *and* $\xleftarrow{>\epsilon} \rtimes \xrightarrow{\epsilon} \subseteq \mathrel{\parallel\!\!\!\rightarrow}$ *hold.*

▶ **Theorem 9** ([32]). *A left-linear TRS is confluent if it is almost parallel closed.*

▶ **Example 10.** Consider the following left-linear and non-terminating TRS, which is a variant of the TRS in [10, Example 5.4].

$$\mathsf{a}(x) \to \mathsf{b}(x) \qquad \mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)) \to \mathsf{g}(\mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)))$$
$$\mathsf{f}(\mathsf{b}(x), y) \to \mathsf{g}(\mathsf{f}(\mathsf{a}(x), y)) \qquad \mathsf{f}(x, \mathsf{b}(y)) \to \mathsf{g}(\mathsf{f}(x, \mathsf{a}(y)))$$

Out of the three critical pairs, two critical pairs including the next diagram (i) are closed by single parallel steps. The remaining pair (ii) joins by performing a single parallel step on each side:

$\mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)) \xrightarrow{\epsilon} \mathsf{g}(\mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)))$, $\mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)) \xrightarrow{1} \mathsf{f}(\mathsf{b}(x), \mathsf{a}(y))$, $\mathsf{f}(\mathsf{b}(x), \mathsf{a}(y)) \mathrel{\parallel\!\!\!\rightarrow} \mathsf{g}(\mathsf{f}(\mathsf{a}(x), \mathsf{a}(y)))$

(i)

$\mathsf{f}(\mathsf{b}(x), \mathsf{b}(y)) \xrightarrow{\epsilon} \mathsf{g}(\mathsf{f}(\mathsf{b}(x), \mathsf{a}(y)))$, $\mathsf{f}(\mathsf{b}(x), \mathsf{b}(y)) \xrightarrow{\epsilon} \mathsf{g}(\mathsf{f}(\mathsf{a}(x), \mathsf{b}(y)))$, $\mathsf{g}(\mathsf{f}(\mathsf{a}(x), \mathsf{b}(y))) \mathrel{\parallel\!\!\!\rightarrow} \mathsf{g}(\mathsf{f}(\mathsf{b}(x), \mathsf{b}(y)))$, $\mathsf{g}(\mathsf{f}(\mathsf{b}(x), \mathsf{a}(y))) \mathrel{\parallel\!\!\!\rightarrow} \mathsf{g}(\mathsf{f}(\mathsf{b}(x), \mathsf{b}(y)))$

(ii)

Thus, the TRS is almost parallel closed. Hence, the TRS is confluent.

Inspired by almost parallel closedness, Gramlich [10] developed a confluence criterion based on *parallel critical pairs* in 1996. Let $t$ be a term and let $P$ be a set of parallel positions in $t$. We write $\mathcal{V}\mathsf{ar}(t, P)$ for the union of $\mathcal{V}\mathsf{ar}(t|_p)$ for all $p \in P$. By $t[u_p]_{p \in P}$ we denote the term that results from replacing in $t$ the subterm at $p$ by a term $u_p$ for all $p \in P$.

▶ **Definition 11.** *Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs, $\ell \to r$ a variant of an $\mathcal{S}$-rule, and $\{\ell_p \to r_p\}_{p \in P}$ a family of variants of $\mathcal{R}$-rules, where $P$ is a set of positions. A local peak*

$$(\ell\sigma)[r_p\sigma]_{p \in P} \;{}_{\mathcal{R}}\!\!\mathrel{\leftarrow\!\!\!\parallel}\; \ell\sigma \xrightarrow{\epsilon}_{\mathcal{S}} r\sigma$$

*is called a* parallel critical peak *between $\mathcal{R}$ and $\mathcal{S}$ if the following conditions hold:*

- *$P \subseteq \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell)$ is a non-empty set of parallel positions in $\ell$,*
- *none of rules $\ell \to r$ and $\ell_p \to r_p$ for $p \in P$ shares a variable with other rules,*
- *$\sigma$ is a most general unifier of $\{\ell_p \approx (\ell|_p)\}_{p \in P}$, and*
- *if $P = \{\epsilon\}$ then $\ell_\epsilon \to r_\epsilon$ is not a variant of $\ell \to r$.*

*When $t \;{}_{\mathcal{R}}\!\!\overset{P}{\mathrel{\leftarrow\!\!\!\parallel}}\; s \xrightarrow{\epsilon}_{\mathcal{S}} u$ is a parallel critical peak, the pair $(t, u)$ is called a* parallel critical pair*, and denoted by $t \;{}_{\mathcal{R}}\!\!\overset{P}{\mathrel{\leftarrow\!\!\!\parallel}} \rtimes \xrightarrow{\epsilon}_{\mathcal{S}} u$. In the case of $P \nsubseteq \{\epsilon\}$ the parallel critical pair is written as $t \;{}_{\mathcal{R}}\!\!\overset{>\epsilon}{\mathrel{\leftarrow\!\!\!\parallel}} \rtimes \xrightarrow{\epsilon}_{\mathcal{S}} u$. Whenever no confusion arises, we abbreviate ${}_{\mathcal{R}}\!\mathrel{\leftarrow\!\!\!\parallel} \rtimes \xrightarrow{\epsilon}_{\mathcal{R}}$ to $\mathrel{\leftarrow\!\!\!\parallel} \rtimes \xrightarrow{\epsilon}$.*

Consider a local peak $t \;{}_{\mathcal{R}}\!\!\overset{P}{\mathrel{\leftarrow\!\!\!\parallel}}\; s \xrightarrow{\epsilon}_{\mathcal{S}} u$ that employs a rule $\ell_p \to r_p$ at $p \in P$ in the left step and a rule $\ell \to r$ in the right step. We say that the peak is *orthogonal* if $P \cap \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell) = \varnothing$. A local peak $t \;{}_{\mathcal{R}}\!\!\xleftarrow{p}\; s \xrightarrow{\epsilon}_{\mathcal{S}} u$ is *orthogonal* if $t \;{}_{\mathcal{R}}\!\!\overset{\{p\}}{\mathrel{\leftarrow\!\!\!\parallel}}\; s \xrightarrow{\epsilon}_{\mathcal{S}} u$ is.

▶ **Theorem 12** ([10]). *A left-linear TRS is confluent if the inclusions $\leftarrow \rtimes \xrightarrow{\epsilon} \subseteq \mathrel{\parallel\!\!\!\rightarrow} \cdot {}^{*}\!\leftarrow$ and $\overset{>\epsilon}{\mathrel{\leftarrow\!\!\!\parallel}} \rtimes \xrightarrow{\epsilon} \subseteq \to^{*}$ hold.*

Unfortunately, this criterion by Gramlich does not subsume (almost) parallel closedness.

▶ **Example 13** (Continued from Example 10)**.** The TRS admits the parallel critical peak $\mathsf{f}(\mathsf{b}(x),\mathsf{b}(y)) \xleftarrow{\{1,2\}}\!\!\!\!+\!\!\!\!+ \mathsf{f}(\mathsf{a}(x),\mathsf{a}(y)) \xrightarrow{\epsilon} \mathsf{g}(\mathsf{f}(\mathsf{a}(x),\mathsf{a}(y)))$. However, $\mathsf{f}(\mathsf{b}(x),\mathsf{b}(y)) \to^* \mathsf{g}(\mathsf{f}(\mathsf{a}(x),\mathsf{a}(y)))$ does not hold.

As noted in the paper [10], Toyama [30] had already obtained in 1981 a closedness result that subsumes Theorem 12. His idea is to impose variable conditions on parallel steps $\mathrel{+\!\!\!\!\!\!\to}$.

▶ **Theorem 14** ([30])**.** *A left-linear TRS is confluent if the following conditions hold:*

**(a)** *The inclusion* $\leftarrow\!\rtimes\!\xrightarrow{\epsilon} \subseteq \mathrel{+\!\!\!\!\!\!\to} \cdot {}^*\!\leftarrow$ *holds.*

**(b)** *For every parallel critical peak* $t \xleftarrow{P}\!\!\!\!+\!\!\!\!+ s \xrightarrow{\epsilon} u$ *there exist a term* $v$ *and a set* $P'$ *of parallel positions such that* $t \to^* v \xleftarrow{P'}\!\!\!\!+\!\!\!\!+ u$ *and* $\mathcal{V}\mathsf{ar}(v,P') \subseteq \mathcal{V}\mathsf{ar}(s,P)$*.*

▶ **Example 15** (Continued from Example 13)**.** The confluence of the TRS in Example 10 can be shown by Theorem 14. Since condition (a) of Theorem 14 follows from the almost parallel closedness, it is enough to verify condition (b). The following parallel critical peak, which Theorem 12 fails to handle, admits the following diagram:

$$\begin{array}{ccc}
\mathsf{f}(\mathsf{a}(x),\mathsf{a}(y)) & \xrightarrow{\epsilon} & \mathsf{g}(\mathsf{f}(\mathsf{a}(x),\mathsf{a}(y))) \\
\{1,2\}\ \big\downarrow\!\!\!+ & & \big\downarrow\!\!\!+\ \{1\cdot 2\} \\
\mathsf{g}(\mathsf{f}(\mathsf{b}(x),\mathsf{b}(y))) & \dashrightarrow & \mathsf{g}(\mathsf{f}(\mathsf{a}(x),\mathsf{b}(y)))
\end{array}$$

Because $\mathcal{V}\mathsf{ar}(\mathsf{g}(\mathsf{f}(\mathsf{a}(x),\mathsf{b}(y))),\{1\cdot 2\}) = \{y\} \subseteq \{x,y\} = \mathcal{V}\mathsf{ar}(\mathsf{f}(\mathsf{a}(x),\mathsf{a}(y)),\{1,2\})$ holds, the parallel critical peak satisfies condition (b) in Theorem 14. Similarly, we can find suitable diagrams for the other parallel critical peaks. Hence, (b) holds for the TRS.

Now we show that Theorem 14 even subsumes Theorem 9. Revisiting the Parallel Moves Lemma [4, Lemma 6.4.4], we show that the variable condition of Theorem 14 is generalized to local peaks of form $\leftarrow\!\!\!\!+\!\!\!\!+ \cdot \xrightarrow{\epsilon}$. We write $\sigma \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} \tau$ if $x\sigma \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} x\tau$ for all variables $x$.

▶ **Lemma 16.** *Let* $\mathcal{R}$ *be a TRS and* $\ell \to r$ *a left-linear rule. Consider a local peak* $\Gamma$ *of the form* $t\ {}_{\mathcal{R}}\!\xleftarrow{P}\!\!\!\!+\!\!\!\!+ s \xrightarrow{\epsilon}_{\{\ell\to r\}} u$*.*

**(a)** *If* $\Gamma$ *is orthogonal,* $t \xrightarrow{\epsilon}_{\{\ell\to r\}} v\ {}_{\mathcal{R}}\!\xleftarrow{P'}\!\!\!\!+\!\!\!\!+ u$ *and* $\mathcal{V}\mathsf{ar}(v,P') \subseteq \mathcal{V}\mathsf{ar}(s,P)$ *for some* $v$ *and* $P'$*.*

**(b)** *Otherwise, there exist a parallel critical peak* $t_0\ {}_{\mathcal{R}}\!\xleftarrow{P_0}\!\!\!\!+\!\!\!\!+ s_0 \xrightarrow{\epsilon}_{\{\ell\to r\}} u_0$ *and substitutions* $\sigma$ *and* $\tau$ *such that* $s = s_0\sigma$*,* $t = t_0\tau$*,* $u = u_0\sigma$*,* $\sigma \mathrel{+\!\!\!\!\!\!\to}_{\mathcal{R}} \tau$*,* $t_0\sigma \xrightarrow{P\setminus P_0}\!\!\!\!\!\!\!\!+\ _{\mathcal{R}}\ t_0\tau$*, and* $P_0 \subseteq P$*.*

**Proof.** As (b) is a known result [40, Lemma 55], we only show (a). Suppose that $\Gamma$ is orthogonal. Since $s \xrightarrow{\epsilon}_{\{\ell\to r\}} u$ holds, there exists a substitution $\sigma$ with $s = \ell\sigma$ and $u = r\sigma$. As $\ell$ is linear and $\Gamma$ is orthogonal, $t = \ell\tau$ and $\sigma \mathrel{+\!\!\!\!\!\!\to} \tau$ for some $\tau$. Take $v = r\tau$ and define $P'$ as follows:

$$P' = \{p_1' \cdot p_2 \mid p_1 \cdot p_2 \in P,\ p_1' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(r), \text{ and } \ell|_{p_1} = r|_{p_1'} \text{ for some } p_1 \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(\ell)\}$$

Clearly, $t \xrightarrow{\epsilon}_{\{\ell\to r\}} v$ holds. So it remains to show $u \xrightarrow{P'}\!\!\!\!\!\!\!+_{\mathcal{R}} v$ and $\mathcal{V}\mathsf{ar}(v,P') \subseteq \mathcal{V}\mathsf{ar}(s,P)$. Let $p'$ be an arbitrary position in $P'$. There exist positions $p_1 \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(\ell)$, $p_1' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(r)$, and $p_2$ such that $p' = p_1' \cdot p_2$, $p_1 \cdot p_2 \in P$, and $\ell|_{p_1} = r|_{p_1'}$. Denoting $p_1 \cdot p_2$ by $p$, we have the identities:

$$u|_{p'} = (r\sigma)|_{p_1'\cdot p_2} = (r|_{p_1'}\sigma)|_{p_2} = (\ell|_{p_1}\sigma)|_{p_2} = (\ell\sigma)|_{p_1\cdot p_2} = s|_p$$
$$v|_{p'} = (r\tau)|_{p_1'\cdot p_2} = (r|_{p_1'}\tau)|_{p_2} = (\ell|_{p_1}\tau)|_{p_2} = (\ell\tau)|_{p_1\cdot p_2} = t|_p$$

From $s \xrightarrow{P}\!\!\!\!\!\!\!\!+\!\!\!+\!\!\!\!\!\rightarrow_{\mathcal{R}} t$ we obtain $s|_p \xrightarrow{\epsilon}_{\mathcal{R}} t|_p$ and thus $u|_{p'} \xrightarrow{\epsilon}_{\mathcal{R}} v|_{p'}$. Therefore, $u \xrightarrow{P'}_{\mathcal{R}} v$ is obtained. Moreover, we have $\mathcal{V}\mathsf{ar}(v|_{p'}) = \mathcal{V}\mathsf{ar}(t|_p) \subseteq \mathcal{V}\mathsf{ar}(s|_p) \subseteq \mathcal{V}\mathsf{ar}(s,P)$. As $\mathcal{V}\mathsf{ar}(v,P')$ is the union of $\mathcal{V}\mathsf{ar}(v|_{p'})$ for all $p' \in P'$, the desired inclusion $\mathcal{V}\mathsf{ar}(v,P') \subseteq \mathcal{V}\mathsf{ar}(s,P)$ follows. ◀

For almost parallel closed TRSs the above statement is extended to local peaks $\nleftarrow\!\!\!\!\!+\!\!\!+ \cdot +\!\!\!+\!\!\!\rightarrow$ of parallel steps. In its proof we measure parallel steps $s \xrightarrow{P} t$ in such a local peak by the *amount of contractums* $|t|_P$, namely the sum of $|(t|_p)|$ for all $p \in P$. Note that this measure attributes to [24, 19].

▶ **Lemma 17.** *Consider a left-linear almost parallel closed TRS. If $t \xleftarrow{P_1} s \xrightarrow{P_2} u$ then*

- *$t \to^* v_1 \xleftarrow{P_1'} u$ for some $v_1$ and $P_1'$ with $\mathcal{V}\mathsf{ar}(v_1,P_1') \subseteq \mathcal{V}\mathsf{ar}(s,P_1)$, and*
- *$t \xrightarrow{P_2'} v_2 \;{}^*\!\!\leftarrow u$ for some $v_2$ and $P_2'$ with $\mathcal{V}\mathsf{ar}(v_2,P_2') \subseteq \mathcal{V}\mathsf{ar}(s,P_2)$.*

**Proof.** Let $\Gamma\colon t \xleftarrow{P_1} s \xrightarrow{P_2} u$ be a local peak. We show the claim by well-founded induction on $(|t|_{P_1} + |u|_{P_2}, s)$ with respect to $\succ$. Here $(m,s) \succ (n,t)$ if either $m > n$, or $m = n$ and $t$ is a proper subterm of $s$. Depending on the shape of $\Gamma$, we distinguish six cases.

1. If $P_1$ or $P_2$ is empty then the claim follows from the fact: $\mathcal{V}\mathsf{ar}(v,P) \subseteq \mathcal{V}\mathsf{ar}(w,P)$ if $w \xrightarrow{P} v$.
2. If $P_1$ or $P_2$ is $\{\epsilon\}$ and $\Gamma$ is orthogonal then Lemma 16(a) applies.
3. If $P_1 = P_2 = \{\epsilon\}$ and $\Gamma$ is not orthogonal then $\Gamma$ is an instance of a critical peak. By almost parallel closedness $t \to^* v_1 \xleftarrow{Q_1} u$ and $t \xrightarrow{Q_2} v_2 \;{}^*\!\!\leftarrow u$ for some $v_1$, $v_2$, $Q_1$, and $Q_2$. For each $k \in \{1,2\}$ we have $s \to^* v_k$, so $\mathcal{V}\mathsf{ar}(v_k) \subseteq \mathcal{V}\mathsf{ar}(s)$ follows. Thus, $\mathcal{V}\mathsf{ar}(v_k,Q_k) \subseteq \mathcal{V}\mathsf{ar}(v_k) \subseteq \mathcal{V}\mathsf{ar}(s) = \mathcal{V}\mathsf{ar}(s,\{\epsilon\})$. The claim holds.
4. If $P_1 \nsubseteq \{\epsilon\}$, $P_2 = \{\epsilon\}$, and $\Gamma$ is not orthogonal then there is $p \in P_1$ such that $s' \xleftarrow{p} s \xrightarrow{\epsilon} u$ is an instance of a critical peak and $s' \xrightarrow{P_1 \setminus \{p\}} t$ follows by Lemma 16(b) where $P = \{p\}$. By the almost parallel closedness $s' \xrightarrow{P_2'} u$ for some $P_2'$. Since $P_2'$ is a set of parallel positions in $u$, we have $|u|_{\{\epsilon\}} = |u| \geqslant |u|_{P_2'}$. As $|u|_{\{\epsilon\}} \geqslant |u|_{P_2'}$ and $|t|_{P_1} > |t|_{P_1 \setminus \{p\}}$ yield $|t|_{P_1} + |u|_{\{\epsilon\}} > |t|_{P_1 \setminus \{p\}} + |u|_{P_2'}$, we obtain the inequality:
   $$(|t|_{P_1} + |u|_{P_2}, s) \succ (|t|_{P_1 \setminus \{p\}} + |u|_{P_2'}, s')$$
   Thus, the claim follows by the induction hypothesis for $t \xleftarrow{P_1 \setminus \{p\}} s' \xrightarrow{P_2'} u$ and the inclusions $\mathcal{V}\mathsf{ar}(s', P_1 \setminus \{p\}) \subseteq \mathcal{V}\mathsf{ar}(s,P_1)$ and $\mathcal{V}\mathsf{ar}(s',P_2') \subseteq \mathcal{V}\mathsf{ar}(s,\{\epsilon\})$.
5. If $P_1 = \{\epsilon\}$, $P_2 \nsubseteq \{\epsilon\}$, and $\Gamma$ is not orthogonal then the proof is analogous to the last case.
6. If $P_1 \nsubseteq \{\epsilon\}$ and $P_2 \nsubseteq \{\epsilon\}$ then we may assume $s = f(s_1,\dots,s_n)$, $t = f(t_1,\dots,t_n)$, $u = f(u_1,\dots,u_n)$, and $t_i \xleftarrow{P_1^i} s_i \xrightarrow{P_2^i} u_i$ for all $1 \leqslant i \leqslant n$. Here $P_k^i$ denotes the set $\{p \mid i \cdot p \in P_k\}$. For each $i \in \{1,\dots,n\}$, we have $|t|_{P_1} \geqslant |t_i|_{P_1^i}$ and $|u|_{P_2} \geqslant |u_i|_{P_2^i}$, and therefore $|t|_{P_1} + |u|_{P_2} \geqslant |t_i|_{P_1^i} + |u_i|_{P_2^i}$. So we deduce the following inequality:
   $$(|t|_{P_1} + |u|_{P_2}, s) \succ (|t_i|_{P_1^i} + |u_i|_{P_2^i}, s_i)$$
   Consider an $i$-th peak $t_i \xleftarrow{P_1^i} s_i \xrightarrow{P_2^i} u_i$. By the induction hypothesis it admits valleys of the forms $t_i \to^* v_1^i \xleftarrow{Q_1^i} u_i$ and $t_i \xrightarrow{Q_2^i} v_2^i \;{}^*\!\!\leftarrow u_i$ such that $\mathcal{V}\mathsf{ar}(v_k^i,Q_k^i) \subseteq \mathcal{V}\mathsf{ar}(s_i,P_k^i)$ for both $k \in \{1,2\}$. For each $k$, define $Q_k = \{i \cdot q \mid 1 \leqslant i \leqslant n \text{ and } q \in Q_k^i\}$ and $v_k = f(v_k^1,\dots,v_k^n)$. Then we have $t \to^* v_1 \xleftarrow{Q_1} u$ and $t \xrightarrow{Q_2} v_2 \;{}^*\!\!\leftarrow u$. Moreover,
   $$\mathcal{V}\mathsf{ar}(v_k,Q_k) = \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(v_k^i,Q_k^i) \subseteq \bigcup_{i=1}^{n} \mathcal{V}\mathsf{ar}(s_i,P_k^i) = \mathcal{V}\mathsf{ar}(s,P_k)$$
   holds. Hence, the claim follows. ◀

▶ **Theorem 18.** *Every left-linear and almost parallel closed TRS satisfies conditions (a) and (b) of Theorem 14. In other words, Theorem 14 subsumes Theorem 9.*

**Proof.** Since (parallel) critical peaks are instances of $\nleftarrow\!\!\!\!\!\!\!\!+ \cdot \dashrightarrow$, Lemma 17 entails the claim. ◀

Note that Theorem 9 does not subsume Theorem 14 as witnessed by the TRS consisting of the four rules $\mathsf{f(a)} \to \mathsf{c}$, $\mathsf{a} \to \mathsf{b}$, $\mathsf{f(b)} \to \mathsf{b}$, and $\mathsf{c} \to \mathsf{b}$. In Section 6 we will see that Theorem 14 is subsumed by a variant of rule labeling.

## 4 Decreasing Diagrams with Commuting Subsystems

We make a variant of decreasing diagrams [35, 37]. First we recall the commutation version of the technique [37]. Let $\mathcal{A} = (A, \{\to_{1,\alpha}\}_{\alpha \in I})$ and $\mathcal{B} = (A, \{\to_{2,\beta}\}_{\beta \in J})$ be $I$-indexed and $J$-indexed ARSs on the same domain, respectively. Let $>$ be a well-founded order $>$ on $I \cup J$. By $\curlyvee\alpha$ we denote the set $\{\beta \in I \cup J \mid \alpha > \beta\}$, and by $\curlyvee\alpha\beta$ we denote $(\curlyvee\alpha) \cup (\curlyvee\beta)$. We say that a local peak $b \;{}_{1,\alpha}\!\leftarrow a \to_{2,\beta} c$ is *decreasing* if

$$b \xleftrightarrow[\curlyvee\alpha]{*} \cdot \xrightarrow[2,\beta]{=} \cdot \xleftrightarrow[\curlyvee\alpha\beta]{*} \cdot \xleftarrow[1,\alpha]{=} \cdot \xleftrightarrow[\curlyvee\beta]{*} c$$

holds. Here $\leftrightarrow_K$ stands for the union of ${}_{1,\gamma}\!\leftarrow$ and $\to_{2,\gamma}$ for all $\gamma \in K$. The ARSs $\mathcal{A}$ and $\mathcal{B}$ are *decreasing* if every local peak $b \;{}_{1,\alpha}\!\leftarrow a \to_{2,\beta} c$ with $(\alpha, \beta) \in I \times J$ is decreasing. In the case of $\mathcal{A} = \mathcal{B}$, we simply say that $\mathcal{A}$ is decreasing.

▶ **Theorem 19** ([37]). *If two ARSs are decreasing then they commute.*

We present the abstract principle of our compositional criteria. The idea of using the minimum index in the decreasing diagram technique is taken from [16, 9, 7].

▶ **Theorem 20.** *Let $\mathcal{A} = (A, \{\to_{1,\alpha}\}_{\alpha \in I})$ and $\mathcal{B} = (A, \{\to_{2,\beta}\}_{\beta \in I})$ be $I$-indexed ARSs equipped with a well-founded order $>$ on $I$. Suppose that $\bot$ is the minimum element in $I$ and $\to_{1,\bot}$ and $\to_{2,\bot}$ commute. The ARSs $\mathcal{A}$ and $\mathcal{B}$ commute if every local peak ${}_{1,\alpha}\!\leftarrow \cdot \to_{2,\beta}$ with $(\alpha, \beta) \in I^2 \setminus \{(\bot, \bot)\}$ is decreasing.*

**Proof.** We define the two ARSs $\mathcal{A}' = (A, \{\Rightarrow_{1,\alpha}\}_{\alpha \in I})$ and $\mathcal{B}' = (A, \{\Rightarrow_{2,\alpha}\}_{\alpha \in I})$ as follows:

$$\Rightarrow_{i,\alpha} = \begin{cases} \to_{i,\alpha}^* & \text{if } \alpha = \bot \\ \to_{i,\alpha} & \text{otherwise} \end{cases}$$

Since $\to_{\mathcal{A}}^* = \Rightarrow_{\mathcal{A}}^*$ and $\to_{\mathcal{B}}^* = \Rightarrow_{\mathcal{B}}^*$, the commutation of $\mathcal{A}$ and $\mathcal{B}$ follows from that of $\mathcal{A}'$ and $\mathcal{B}'$. We show the latter by proving decreasingness of $\mathcal{A}'$ and $\mathcal{B}'$ with respect to the given well-founded order $>$. Let $\Gamma$ be a local peak of form ${}_{1,\alpha}\!\Leftarrow \cdot \Rightarrow_{2,\beta}$. We distinguish four cases.

- If neither $\alpha$ nor $\beta$ is $\bot$ then decreasingness of $\Gamma$ follows from the assumption.
- If both $\alpha$ and $\beta$ are $\bot$ then the commutation of $\to_{1,\bot}$ and $\to_{2,\bot}$ yields the inclusion:

  $$\xLeftarrow[1,\bot]{} \cdot \xRightarrow[2,\bot]{} \subseteq \xRightarrow[2,\bot]{} \cdot \xLeftarrow[1,\bot]{}$$

  Thus $\Gamma$ is decreasing.
- If $\beta > \alpha = \bot$ then we have ${}_{1,\alpha}\!\leftarrow \cdot \to_{2,\beta} \subseteq \to_{2,\beta}^{=} \cdot \leftrightarrow_{\curlyvee\beta}^*$ Therefore, easy induction on $n$ shows the inclusion ${}_{1,\alpha}^{\;n}\!\leftarrow \cdot \to_{2,\beta} \subseteq \to_{2,\beta}^{=} \cdot \leftrightarrow_{\curlyvee\beta}^*$ for all $n \in \mathbb{N}$. Thus,

  $$\xLeftarrow[1,\alpha]{} \cdot \xRightarrow[2,\beta]{} = \xleftarrow[1,\alpha]{*} \cdot \xrightarrow[2,\beta]{} \subseteq \xrightarrow[2,\beta]{=} \cdot \xleftrightarrow[\curlyvee\beta]{*} = \xRightarrow[2,\beta]{=} \cdot \xLeftrightarrow[\curlyvee\beta]{*}$$

  holds, where $\Leftrightarrow_J$ stands for ${}_{1,J}\!\Leftarrow \cup \Rightarrow_{2,J}$. Hence $\Gamma$ is decreasing.
- The case that $\alpha > \beta = \bot$ is analogous to the last case. ◀

## 5 Orthogonality

As a first example of compositional confluence criteria and its derivation, we pick up Rosen's confluence criterion by orthogonality [26]. *Orthogonal* TRSs are left-linear TRSs having no critical pairs. Their confluence property can be shown by decreasingness of parallel steps. We briefly recall its proof. Left-linear TRSs are *mutually orthogonal* if ${}_{\mathcal{R}}{\leftarrow}\!\rtimes\!\xrightarrow{\epsilon}_{\mathcal{S}} = \varnothing$ and ${}_{\mathcal{S}}{\leftarrow}\!\rtimes\!\xrightarrow{\epsilon}_{\mathcal{R}} = \varnothing$. Note that orthogonality of $\mathcal{R}$ and mutual orthogonality of $\mathcal{R}$ and $\mathcal{R}$ are equivalent.

▶ **Lemma 21** ([4, Theorem 9.3.11])**.** *For mutually orthogonal TRSs $\mathcal{R}$ and $\mathcal{S}$ the inclusion ${}_{\mathcal{R}}{\nleftarrow} \cdot \nrightarrow_{\mathcal{S}} \subseteq \nrightarrow_{\mathcal{S}} \cdot {}_{\mathcal{R}}{\nleftarrow}$ holds.*

▶ **Theorem 22** ([26])**.** *Every orthogonal TRS $\mathcal{R}$ is confluent.*

**Proof.** Let $\mathcal{A} = (\mathcal{T}(\mathcal{F},\mathcal{V}), \{\nrightarrow_1\})$ be the ARS equipped with the empty order $>$ on $\{1\}$, where $\nrightarrow_1 = \nrightarrow_{\mathcal{R}}$. According to Lemma 5 and Theorem 19, it is enough to show that $\mathcal{A}$ is decreasing. Since Lemma 21 yields ${}_1{\nleftarrow} \cdot \nrightarrow_1 \subseteq \nrightarrow_1 \cdot {}_1{\nleftarrow}$, the decreasingness of $\mathcal{A}$ follows. ◀

The theorem can be recast as a compositional criterion that uses a confluent subsystem $\mathcal{C}$ of a given TRS $\mathcal{R}$. For this sake we switch the underlying criterion from Theorem 19 to Theorem 20, setting the relation of the minimum index $\bot$ to $\nrightarrow_{\mathcal{C}}$.

▶ **Theorem 23.** *A left-linear TRS $\mathcal{R}$ is confluent if $\mathcal{R}$ and $\mathcal{R} \setminus \mathcal{C}$ are mutually orthogonal for some confluent TRS $\mathcal{C}$ with $\mathcal{C} \subseteq \mathcal{R}$.*

**Proof.** Let $\mathcal{A} = (\mathcal{T}(\mathcal{F},\mathcal{V}), \{\nrightarrow_0, \nrightarrow_1\})$ be the ARS equipped with the well-founded order $1 > 0$, where $\nrightarrow_0 = \nrightarrow_{\mathcal{C}}$ and $\nrightarrow_1 = \nrightarrow_{\mathcal{R}\setminus\mathcal{C}}$. According to Lemma 5 and Theorem 19, it is enough to show that $\mathcal{A}$ is decreasing. Since $\mathcal{R}$ and $\mathcal{R} \setminus \mathcal{C}$ are mutually orthogonal, Lemma 21 yields ${}_k{\nleftarrow} \cdot \nrightarrow_m \subseteq \nrightarrow_m \cdot {}_k{\nleftarrow}$ for all $(k,m) \in \{0,1\}^2 \setminus \{(0,0)\}$, from which the decreasingness of $\mathcal{A}$ follows. Hence, Theorem 20 applies. ◀

We can derive a more general criterion by exploiting the flexible valley form of decreasing diagrams. We will adopt parallel critical pairs. It causes no loss of confluence proving power of Theorem 23 as ${}_{\mathcal{R}}{\nleftarrow}\!\rtimes\!\xrightarrow{\epsilon}_{\mathcal{S}} = \varnothing$ is equivalent to ${}_{\mathcal{R}}{\leftarrow}\!\rtimes\!\xrightarrow{\epsilon}_{\mathcal{S}} = \varnothing$.

▶ **Theorem 24.** *A left-linear TRS $\mathcal{R}$ is confluent if ${}_{\mathcal{R}}{\nleftarrow}\!\rtimes\!\xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \leftrightarrow^*_{\mathcal{C}}$ holds for some confluent TRS $\mathcal{C}$ with $\mathcal{C} \subseteq \mathcal{R}$.*

**Proof.** Recall the ARS used in Theorem 23. According to Lemma 5 and Theorem 20, it is sufficient to show that every local peak

$$\Gamma : t \underset{k}{\overset{P}{\nleftarrow}} s \underset{m}{\overset{Q}{\nrightarrow}} u$$

with $(k,m) \neq (0,0)$ is decreasing. To this end, we show $t \nrightarrow_m \cdot \nleftrightarrow^*_0 \cdot {}_k{\nleftarrow}\, u$ by structural induction on $s$. Depending on the shape of $\Gamma$, we distinguish four cases.

1. If $P$ or $Q$ is empty then the claim is trivial.
2. If $P$ or $Q$ is $\{\epsilon\}$ and $\Gamma$ is orthogonal then Lemma 16(a) yields the join form $t \nrightarrow_m \cdot {}_k{\nleftarrow}\, u$.
3. If $P \neq \varnothing$, $Q = \{\epsilon\}$, and $\Gamma$ is not orthogonal then by Lemma 16(b) there exist a parallel critical peak $t_0 \; {}_k{\nleftarrow}\; s_0 \xrightarrow{\epsilon}_m u_0$ and substitutions $\sigma$ and $\tau$ such that $s = s_0\sigma$, $t = t_0\tau$, $u = u_0\sigma$, and $\sigma \nrightarrow_k \tau$. The assumption $t_0 \leftrightarrow^*_{\mathcal{C}} u_0$ yields $t_0\tau \nleftrightarrow^*_0 u_0\tau$. Therefore, $t = t_0\tau \nleftrightarrow^*_0 u_0\tau \; {}_k{\nleftarrow}\; u_0\sigma = u$ follows.

**4.** If $P = \{\epsilon\}$, $Q \neq \varnothing$, and $\Gamma$ is not orthogonal then the proof is analogous to the last case.
**5.** If $P \nsubseteq \{\epsilon\}$ and $Q \nsubseteq \{\epsilon\}$ then $s$, $t$, and $u$ can be written as $f(s_1, \ldots, s_n)$, $f(t_1, \ldots, t_n)$, and $f(u_1, \ldots, u_n)$ respectively, and moreover, $t_i \ {}_k{\leftarrow\!\!\!\!\!\!\;+\!\!\!\!+} \ s_i \twoheadrightarrow\!\!\!\!\!\!+_m u_i$ holds for all $1 \leqslant i \leqslant n$. For every $i$ the induction hypothesis yields $t_i \twoheadrightarrow\!\!\!\!\!\!+_m v_i \leftrightarrow\!\!\!\!\!\!+^*_0 w_i \ {}_k{\leftarrow\!\!\!\!\!\!+} \ u_i$ for some $v_i$ and $w_i$. Therefore, the desired conversion $t \twoheadrightarrow\!\!\!\!\!\!+_m v \leftrightarrow\!\!\!\!\!\!+^*_0 w \ {}_k{\leftarrow\!\!\!\!\!\!+} \ u$ holds for $v = f(v_1, \ldots, v_n)$ and $w = f(w_1, \ldots, w_n)$. ◀

From Takahashi's proposition [28] (see also [29, Proposition 9.3.5]) we can deduce that ${}_{\mathcal{R}}{\leftarrow\!\!\!\!\!\!+} \rtimes \xrightarrow{\epsilon}_{\mathcal{R}} \subseteq =$ is equivalent to ${}_{\mathcal{R}}{\leftarrow} \rtimes \xrightarrow{\epsilon}_{\mathcal{R}} \subseteq =$. Thus, Theorem 24 subsumes Theorem 23. Note that when $\mathcal{C} = \varnothing$, Theorem 24 simulates the weak orthogonality criterion.

▶ **Example 25.** By successive application of Theorem 24 we show the confluence of the left-linear TRS $\mathcal{R}$ (COPS [13] number 62), taken from [25]:

$$\begin{array}{rrrrrr}
1{:} & x - 0 \to x & 7{:} & \mathsf{gcd}(x, 0) \to x & 13{:} & \mathsf{if}(\mathsf{true}, x, y) \to x \\
2{:} & 0 - x \to 0 & 8{:} & \mathsf{gcd}(0, x) \to x & 14{:} & \mathsf{if}(\mathsf{false}, x, y) \to y \\
3{:} & \mathsf{s}(x) - \mathsf{s}(y) \to x - y & 9{:} & \mathsf{gcd}(x, y) \to \mathsf{gcd}(y, \mathsf{mod}(x, y)) & & \\
4{:} & x < 0 \to \mathsf{false} & 10{:} & \mathsf{mod}(x, 0) \to x & & \\
5{:} & 0 < \mathsf{s}(y) \to \mathsf{true} & 11{:} & \mathsf{mod}(0, y) \to 0 & & \\
6{:} & \mathsf{s}(x) < \mathsf{s}(y) \to x < y & 12{:} & \mathsf{mod}(x, \mathsf{s}(y)) \to \mathsf{if}(x < \mathsf{s}(y), x, \mathsf{mod}(x - \mathsf{s}(y), \mathsf{s}(y))) & &
\end{array}$$

Let $\mathcal{C} = \{5, 7, 8, 10, 11, 13\}$. The six non-trivial parallel critical pairs of $\mathcal{R}$ are

$$(x, \mathsf{gcd}(0, \mathsf{mod}(x, 0))) \qquad (y, \mathsf{gcd}(y, \mathsf{mod}(0, y))) \qquad (0, \mathsf{if}(0 < \mathsf{s}(y), 0, \mathsf{mod}(0 - \mathsf{s}(y), \mathsf{s}(y))))$$

and their symmetric versions. All of them are joinable by $\mathcal{C}$. So it remains to show that $\mathcal{C}$ is confluent. Because $\mathcal{C}$ only admits trivial parallel critical pairs, ${}_{\mathcal{C}}{\leftarrow\!\!\!\!\!\!+} \rtimes \xrightarrow{\epsilon}_{\mathcal{C}} \subseteq \leftrightarrow^*_{\varnothing}$ holds. Therefore, the confluence of $\mathcal{C}$ is concluded if we show the confluence of the empty system. The latter claim is trivial. This completes the proof.

Theorem 24 is a generalization of Toyama's yet another theorem:

▶ **Corollary 26** ([33]). *A left-linear TRS $\mathcal{R}$ is confluent if ${}_{\mathcal{R}}{\leftarrow\!\!\!\!\!\!+} \rtimes \xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \leftrightarrow^*_{\mathcal{C}}$ holds for some terminating and confluent TRS $\mathcal{C}$ with $\mathcal{C} \subseteq \mathcal{R}$.*

## 6 Rule Labeling

In this section we recast the *rule labeling* criterion [37, 40, 7] in a compositional form. Rule labeling is a direct application of decreasing diagrams to confluence proofs for TRSs. It labels rewrite steps by their employed rewrite rules and compares indexes of them. Among others, we focus on the variant of rule labeling based on parallel critical pairs, introduced by Zankl et al. [40].

▶ **Definition 27.** *Let $\mathcal{R}$ be a TRS. A* labeling function *for $\mathcal{R}$ is a function from $\mathcal{R}$ to $\mathbb{N}$. Given a labeling function $\phi$ and a number $k \in \mathbb{N}$, we define the TRS $\mathcal{R}_{\phi,k}$ as follows:*

$$\mathcal{R}_{\phi,k} = \{\ell \to r \in \mathcal{R} \mid \phi(\ell \to r) \leqslant k\}$$

*The relations $\to_{\mathcal{R}_{\phi,k}}$ and $\twoheadrightarrow\!\!\!\!\!\!+_{\mathcal{R}_{\phi,k}}$ are abbreviated to $\to_{\phi,k}$ and $\twoheadrightarrow\!\!\!\!\!\!+_{\phi,k}$. Let $\phi$ and $\psi$ be labeling functions for $\mathcal{R}$. We say that a local peak $t \xleftarrow[\phi,k]{P}\!\!\!\!\!\!+ \ s \xrightarrow[\psi,m]{\epsilon} u$ is* $(\psi, \phi)$-decreasing *if*

$$t \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{}\!\!\!\!\!\!\!\!+\ \cdot \xleftrightarrow[\curlyvee km]{*} v \xleftarrow[\phi,k]{P'}\!\!\!\!\!\!\!\!+\ \cdot \xleftrightarrow[\curlyvee m]{*} u$$

*and $\mathcal{V}\mathsf{ar}(v, P') \subseteq \mathcal{V}\mathsf{ar}(s, P)$ for some set $P'$ of parallel positions and term $v$. Here $\leftrightarrow_K$ stands for the union of ${}_{\phi,k}{\leftarrow}$ and $\to_{\psi,k}$ for all $k \in K$.*

The following theorem is a commutation-based version of the rule labeling method [40, Theorem 56].

▶ **Theorem 28.** *Let $\mathcal{R}$ be a left-linear TRS, and $\phi$ and $\psi$ its labeling functions. The TRS $\mathcal{R}$ is confluent if the following conditions hold for all $k, m \in \mathbb{N}$.*

- *Every parallel critical peak of form $t \underset{\phi,k}{\leftarrow\!\!\!+\!\!\!+\!\!\!-} s \xrightarrow[\psi,m]{\epsilon} u$ is $(\psi, \phi)$-decreasing.*
- *Every parallel critical peak of form $t \underset{\psi,m}{\leftarrow\!\!\!+\!\!\!+\!\!\!-} s \xrightarrow[\phi,k]{\epsilon} u$ is $(\phi, \psi)$-decreasing.*

With a small example we illustrate the usage of rule labeling.

▶ **Example 29.** Consider the left-linear TRS $\mathcal{R}$:

$$(x + y) + z \to x + (y + z) \qquad\qquad x + (y + z) \to (x + y) + z$$

We define the labeling functions $\phi$ and $\psi$ as follows: $\phi(\ell \to r) = 0$ and $\psi(\ell \to r) = 1$ for all $\ell \to r \in \mathcal{R}$. Because $\mathcal{R}$ is reversible, all parallel critical peaks can be closed by $\to_{\phi,0}$-steps, like the following diagram:

$$\begin{array}{ccccc} s = ((x+y)+z)+w & & \xrightarrow[\psi,1]{\epsilon} & & (x+y)+(z+w) \\ \{1\} \Big\downarrow\!\!\!\!+\!\!\!\!+ \ \phi,0 & & & & \varnothing \ \Big\downarrow\!\!\!\!+\!\!\!\!+ \ \phi,0 \\ (x+(y+z))+w & \underset{\phi,0}{\dashrightarrow} & ((x+y)+z)+w & \underset{\phi,0}{\dashleftarrow} & (x+y)+(z+w) = v \end{array}$$

As $\mathcal{V}\mathsf{ar}(v, \varnothing) = \varnothing \subseteq \{x, y, z\} = \mathcal{V}\mathsf{ar}(s, \{1\})$, this parallel critical peak is $(\psi, \phi)$-decreasing. In a similar way the other peaks can also be verified. Hence, the TRS $\mathcal{R}$ is confluent.

We make the rule labeling compositional. The following lemma is used for composing parallel steps.

▶ **Lemma 30** ([40, Lemma 51(b)])**.** *If $s \overset{P}{\twoheadrightarrow\!\!\!\!+}_{\mathcal{R}} t$, $\sigma \twoheadrightarrow\!\!\!\!+_{\mathcal{R}} \tau$, and $x\sigma = x\tau$ for all $x \in \mathcal{V}\mathsf{ar}(s, P)$ then $s\sigma \twoheadrightarrow\!\!\!\!+_{\mathcal{R}} t\tau$.*

The next theorem is a compositional version of the rule labeling criterion. Note that by taking $\mathcal{C} := \mathcal{R}_{\phi,0} = \mathcal{R}_{\psi,0}$ it can be used as a compositional confluence criterion parameterized by $\mathcal{C}$.

▶ **Theorem 31.** *Let $\mathcal{R}$ be a left-linear TRS, and $\phi$ and $\psi$ its labeling functions. Suppose that $\mathcal{R}_{\phi,0}$ and $\mathcal{R}_{\psi,0}$ commute. The TRS $\mathcal{R}$ is confluent if the following conditions hold for all $(k, m) \in \mathbb{N}^2 \setminus \{(0, 0)\}$.*

- *Every parallel critical peak of form $t \underset{\phi,k}{\leftarrow\!\!\!+\!\!\!+\!\!\!-} s \xrightarrow[\psi,m]{\epsilon} u$ is $(\psi, \phi)$-decreasing.*
- *Every parallel critical peak of form $t \underset{\psi,m}{\leftarrow\!\!\!+\!\!\!+\!\!\!-} s \xrightarrow[\phi,k]{\epsilon} u$ is $(\phi, \psi)$-decreasing.*

**Proof.** Consider the ARSs $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \{\twoheadrightarrow\!\!\!\!+_{\phi,k}\}_{k\in\mathbb{N}})$ and $(\mathcal{T}(\mathcal{F}, \mathcal{V}), \{\twoheadrightarrow\!\!\!\!+_{\psi,m}\}_{m\in\mathbb{N}})$. According to Lemma 5 and Theorem 20, it is sufficient to show that every local peak

$$\Gamma\colon t \underset{\phi,k}{\overset{P}{\leftarrow\!\!\!+\!\!\!+\!\!\!-}} s \underset{\psi,m}{\overset{Q}{-\!\!\!+\!\!\!+\!\!\!\rightarrow}} u$$

with $(k, m) \neq (0, 0)$ is decreasing. To this end, we perform structural induction on $s$. Depending on the shape of $\Gamma$, we distinguish five cases.

1. If $P$ or $Q$ is empty then the claim is trivial.
2. If $P$ or $Q$ is $\{\epsilon\}$ and $\Gamma$ is orthogonal then Lemma 16(a) yields the join form $t \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftarrow[\phi,k]{\shortparallel} u$.
3. If $P \neq \varnothing$, $Q = \{\epsilon\}$, and $\Gamma$ is not orthogonal then by Lemma 16(b) there exist a parallel critical peak $t_0 \xleftarrow[\phi,k]{P_1} s_0 \xrightarrow[\psi,m]{\epsilon} u_0$ and substitutions $\sigma$ and $\tau$ such that $t = t_0\tau$, $u = u_0\sigma$, $\sigma \twoheadrightarrow \tau$, $t_0\sigma \xrightarrow{P \setminus P_1}_{\mathcal{R}} t_0\tau$, and $P_1 \subseteq P$. Since $\leftrightarrow\!\!\!\shortparallel_J^* = \leftrightarrow_J^*$ holds in general, the assumption yields

   $$t_0 \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftrightarrow[\curlyvee km]{*} v_0 \xleftarrow[\phi,k]{P_1'} w_0 \xleftrightarrow[\curlyvee m]{*} u_0$$

   and $\mathcal{V}\mathsf{ar}(v_0, P_1') \subseteq \mathcal{V}\mathsf{ar}(s_0, P_1)$ for some $v_0$, $w_0$, and $P_1'$. Since the rewrite steps are closed under substitutions, the following relations are obtained:

   $$t_0\tau \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftrightarrow[\curlyvee km]{*} v_0\tau \qquad\qquad w_0\sigma \xleftrightarrow[\curlyvee m]{*} u_0\sigma$$

   Since $t_0\sigma|_p = t_0\tau|_p$ holds for all $p \in P_1$, the identity $x\sigma = x\tau$ holds for all $x \in \mathcal{V}\mathsf{ar}(s_0, P_1)$. Therefore, $x\sigma = x\tau$ holds for all $x \in \mathcal{V}\mathsf{ar}(v_0, P_1')$. Because $v_0 \xrightarrow[\phi,k]{P_1'} v_0$, $\sigma \twoheadrightarrow \tau$, and $x\sigma = x\tau$ for all $x \in \mathcal{V}\mathsf{ar}(v_0, P_1')$ hold, Lemma 30 yields $w_0\sigma \xrightarrow[\phi,k]{\shortparallel} v_0\tau$. Hence, the decreasingness of $\Gamma$ is witnessed by the following sequence:

   $$t = t_0\tau \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftrightarrow[\curlyvee km]{*} v_0\tau \xleftarrow[\phi,k]{\shortparallel} w_0\sigma \xleftrightarrow[\curlyvee m]{*} u_0\sigma = u$$

   Note that the construction is depicted in Figure 1.
4. If $P = \{\epsilon\}$, $Q \neq \varnothing$, and $\Gamma$ is not orthogonal then the proof is analogous to the last case.
5. If $P \nsubseteq \{\epsilon\}$ and $Q \nsubseteq \{\epsilon\}$ then $s$, $t$, and $u$ can be written as $f(s_1, \ldots, s_n)$, $f(t_1, \ldots, t_n)$, and $f(u_1, \ldots, u_n)$ respectively, and moreover, $t_i \xleftarrow[\phi,k]{\shortparallel} s_i \xrightarrow[\psi,m]{\shortparallel} u_i$ holds for all $1 \leqslant i \leqslant n$. By the induction hypotheses we have $t_i \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftrightarrow[\curlyvee km]{*} \cdot \xleftarrow[\phi,k]{\shortparallel} \cdot \xleftrightarrow[\curlyvee m]{*} u_i$ for all $1 \leqslant i \leqslant n$. Therefore, we obtain the desired relations:

   $$t = f(t_1, \ldots, t_n) \xleftrightarrow[\curlyvee k]{*} \cdot \xrightarrow[\psi,m]{\shortparallel} \cdot \xleftrightarrow[\curlyvee km]{*} \cdot \xleftarrow[\phi,k]{\shortparallel} \cdot \xleftrightarrow[\curlyvee m]{*} f(u_1, \ldots, u_n) = u$$

   Hence $\Gamma$ is decreasing. ◀

The original version of rule labeling (Theorem 28) is a special case of Theorem 31: Suppose that labeling functions $\phi$ and $\psi$ for a left-linear TRS $\mathcal{R}$ satisfy the conditions of Theorem 28. By taking the labeling functions $\phi'$ and $\psi'$ with

$$\phi'(\ell \to r) = \phi(\ell \to r) + 1 \qquad\qquad \psi'(\ell \to r) = \psi(\ell \to r) + 1$$

Theorem 31 applies for $\phi'$, $\psi'$, and the empty TRS $\mathcal{C}$.

The next example shows the combination of our rule labeling variant (Theorem 31) with Knuth–Bendix' criterion (Theorem 3).

▶ **Example 32.** Consider the left-linear TRS $\mathcal{R}$:

$$1\colon\ 0 + x \to x \qquad 2\colon\ (x + y) + z \to x + (y + z) \qquad 3\colon\ x + (y + z) \to (x + y) + z$$

Let $\mathcal{C} = \{1, 2\}$. We define the labeling functions $\phi$ and $\psi$ as follows:

$$\phi(\ell \to r) = \psi(\ell \to r) = \begin{cases} 0 & \text{if } \ell \to r \in \mathcal{C} \\ 1 & \text{otherwise} \end{cases}$$

For instance, the parallel critical pairs involving rule 3 admit the following diagrams:

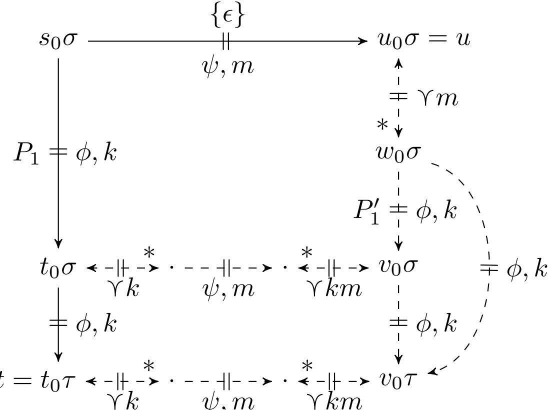



**Figure 1** Proof of Theorem 31.

$$\begin{array}{ccc} x + (0 + z) & \xrightarrow[\psi,1]{\epsilon} & (x + 0) + z \\ \{2\} \Downarrow \phi, 0 & & \phi, 0 \\ x + z & \xleftarrow[\phi,0]{} & x + (0 + z) \end{array} \qquad \begin{array}{ccc} x + (y + (z + w)) & \xrightarrow[\psi,1]{\epsilon} & (x + y) + (z + w) \\ \{2\} \Downarrow \phi, 1 & & \phi, 0 \\ x + ((y + z) + w) & \xleftarrow[\phi,1]{\{2\}} & x + (y + (z + w)) \end{array}$$

They fit for the conditions of Theorem 31. The other parallel critical pairs also admit suitable diagrams. Therefore, it remains to show that $\mathcal{C}$ is confluent. Since $\mathcal{C}$ is terminating and all its critical pairs are joinable, confluence of $\mathcal{C}$ follows by Knuth and Bendix' criterion (Theorem 3). Thus, $\mathcal{R}_{\phi,0}$ and $\mathcal{R}_{\psi,0}$ commute because $\mathcal{R}_{\phi,0} = \mathcal{R}_{\psi,0} = \mathcal{C}$. Hence, by Theorem 31 we conclude that $\mathcal{R}$ is confluent.

While a proof for Theorem 24 is given in Section 3, here we present an alternative proof based on Theorem 31.

**Proof of Theorem 24.** Define the labeling functions $\phi$ and $\psi$ as in Example 32. Then Theorem 31 applies. ◀

We conclude the section by stating that rule labeling based on parallel critical pairs (Theorem 28) subsumes parallel closedness based on parallel critical pairs (Theorem 14): Suppose that conditions (a,b) of Theorem 14 hold. We define $\phi$ and $\psi$ as the constant rule labeling functions $\phi(\ell \to r) = 1$ and $\psi(\ell \to r) = 0$. By using structural induction as well as Lemmata 16 and 30 we can prove the implication

$$t \xleftarrow[\phi,1]{P_1} s \xrightarrow[\psi,0]{} u \implies t \xrightarrow[\psi,0]{*} v \xleftrightarrow[\phi,1]{P_1'} u \text{ and } \mathcal{V}\mathsf{ar}(v, P_1') \subseteq \mathcal{V}\mathsf{ar}(s, P_1) \text{ for some } P_1'$$

Thus, the conditions of Theorem 28 follow. As a consequence, our compositional version (Theorem 31) is also a generalization of parallel closedness.

## 7 Critical Pair Systems

The last example of compositional criteria is a variant of the confluence criterion by critical pair systems [11]. It is known that the original criterion is a generalization of the orthogonal criterion (Theorem 22) and Knuth and Bendix' criterion (Theorem 3) for left-linear TRSs.

▶ **Definition 33.** *The* critical pair system $\mathsf{CPS}(\mathcal{R})$ *of a TRS* $\mathcal{R}$ *is defined as the TRS:*

$$\{s \to t, s \to u \mid t \mathrel{{}_{\mathcal{R}}{\leftarrow}} s \xrightarrow{\epsilon}_{\mathcal{R}} u \text{ is a critical peak}\}$$

▶ **Theorem 34** ([11])**.** *A left-linear and locally confluent TRS* $\mathcal{R}$ *is confluent if* $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ *is terminating (i.e.,* $\mathsf{CPS}(\mathcal{R})$ *is relatively terminating with respect to* $\mathcal{R}$*).*

The theorem is shown by using the decreasing diagram technique (Theorem 19), see [11].

▶ **Example 35.** Consider the left-linear and non-terminating TRS $\mathcal{R}$:

$$\mathsf{s}(\mathsf{p}(x)) \to \mathsf{p}(\mathsf{s}(x)) \qquad \mathsf{p}(\mathsf{s}(x)) \to x \qquad \infty \to \mathsf{s}(\infty)$$

The TRS $\mathcal{R}$ admits two critical pairs and they are joinable:

$$\mathsf{s}(x) \leftarrow \mathsf{s}(\mathsf{p}(\mathsf{s}(x))) \xrightarrow{\epsilon} \mathsf{p}(\mathsf{s}(\mathsf{s}(x))), \quad \mathsf{p}(\mathsf{s}(\mathsf{s}(x))) \dashrightarrow \mathsf{s}(x)$$

$$\mathsf{p}(\mathsf{p}(\mathsf{s}(x))) \leftarrow \mathsf{p}(\mathsf{s}(\mathsf{p}(x))) \xrightarrow{\epsilon} \mathsf{p}(x), \quad \mathsf{p}(\mathsf{p}(\mathsf{s}(x))) \dashrightarrow \mathsf{p}(x)$$

The critical pair system $\mathsf{CPS}(\mathcal{R})$ consists of the four rules:

$$\mathsf{s}(\mathsf{p}(\mathsf{s}(x))) \to \mathsf{s}(x) \qquad \mathsf{p}(\mathsf{s}(\mathsf{p}(x))) \to \mathsf{p}(\mathsf{p}(\mathsf{s}(x)))$$
$$\mathsf{s}(\mathsf{p}(\mathsf{s}(x))) \to \mathsf{p}(\mathsf{s}(\mathsf{s}(x))) \qquad \mathsf{p}(\mathsf{s}(\mathsf{p}(x))) \to \mathsf{p}(x)$$

Termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ can be shown by, e.g., the termination tool $\mathsf{NaTT}$ [38]. Hence the confluence of $\mathcal{R}$ follows by Theorem 34.

We argue about the parallel critical pair version of $\mathsf{CPS}(\mathcal{R})$:

$$\mathsf{PCPS}(\mathcal{R}) = \{s \to t, s \to u \mid t \mathrel{{}_{\mathcal{R}}{\twoheadleftarrow}} s \xrightarrow{\epsilon}_{\mathcal{R}} u \text{ is a parallel critical peak}\}$$

Interestingly, replacing $\mathsf{CPS}(\mathcal{R})$ by $\mathsf{PCPS}(\mathcal{R})$ in Theorem 34 results in the same criterion (see [40]). Since $\to_{\mathsf{CPS}(\mathcal{R})} \subseteq \to_{\mathsf{PCPS}(\mathcal{R})} \subseteq \to_{\mathsf{CPS}(\mathcal{R})} \cdot \twoheadrightarrow_{\mathcal{R}}$ holds, $\to_{\mathsf{CPS}(\mathcal{R})/\mathcal{R}} = \to_{\mathsf{PCPS}(\mathcal{R})/\mathcal{R}}$ follows. So the termination of $\mathsf{PCPS}(\mathcal{R})/\mathcal{R}$ is equivalent to that of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$. However, a compositional form of Theorem 34 may benefit from the use of parallel critical pairs, as seen in Section 5.

▶ **Definition 36.** *Let* $\mathcal{R}$ *and* $\mathcal{C}$ *be TRSs. The* parallel critical pair system $\mathsf{PCPS}(\mathcal{R},\mathcal{C})$ *of* $\mathcal{R}$ *modulo* $\mathcal{C}$ *is defined as the TRS:*

$$\{s \to t, s \to u \mid t \mathrel{{}_{\mathcal{R}}{\twoheadleftarrow}} s \xrightarrow{\epsilon}_{\mathcal{R}} u \text{ is a parallel critical peak but not } t \leftrightarrow^*_{\mathcal{C}} u\}$$

Note that $\mathsf{PCPS}(\mathcal{R},\varnothing) \subseteq \mathsf{PCPS}(\mathcal{R})$ holds in general, and $\mathsf{PCPS}(\mathcal{R},\varnothing) \subsetneq \mathsf{PCPS}(\mathcal{R})$ when $\mathcal{R}$ admits a trivial critical pair. The next lemma relates $\mathsf{PCPS}(\mathcal{R},\mathcal{C})$ to closing forms of parallel critical peaks.

▶ **Lemma 37.** *Let* $\mathcal{R}$ *be a left-linear TRS and* $\mathcal{R}_1$*,* $\mathcal{R}_2$*, and* $\mathcal{C}$ *subsets of* $\mathcal{R}$*, and let* $\mathcal{P} = \mathsf{PCPS}(\mathcal{R},\mathcal{C})$*. Suppose that* ${}_{\mathcal{R}}{\twoheadleftarrow} \rtimes \xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \to^*_{\mathcal{R}} \cdot {}^*_{\mathcal{R}}{\leftarrow}$ *holds. If* $t \mathrel{{}_{\mathcal{R}_1}{\twoheadleftarrow}} s \twoheadrightarrow_{\mathcal{R}_2} u$ *then*

**(i)** $t \twoheadrightarrow_{\mathcal{R}_2} \cdot \leftrightarrow^*_{\mathcal{C}} \cdot {}_{\mathcal{R}_1}{\twoheadleftarrow}\, u$*, or*

**(ii)** $t \mathrel{{}_{\mathcal{R}_1}{\twoheadleftarrow}} t' \mathrel{{}_{\mathcal{P}}{\leftarrow}} s \to_{\mathcal{P}} u' \twoheadrightarrow_{\mathcal{R}_2} u$ *and* $t' \to^*_{\mathcal{R}} \cdot {}^*_{\mathcal{R}}{\leftarrow}\, u'$ *for some* $t'$ *and* $u'$*.*

**Proof.** Let $\Gamma\colon t \mathrel{{}_{\mathcal{R}_1}{\overset{P}{\twoheadleftarrow}}} s \overset{Q}{\twoheadrightarrow}_{\mathcal{R}_2} u$ be a local peak. We use structural induction on $s$. Depending on the form of $\Gamma$, we distinguish five cases.

1. If $P$ or $Q$ is the empty then (i) holds trivially.
2. If $P$ or $Q$ is $\{\epsilon\}$ and $\Gamma$ is orthogonal then (i) follows by Lemma 16(a).
3. If $P \neq \varnothing$, $Q = \{\epsilon\}$, and $\Gamma$ is not orthogonal then we distinguish two cases.
   - If there exist $P_0$, $t_0$, $u_0$, and $\sigma$ such that "$P_0 \subseteq P$, $t \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow t_0\sigma \;{}_{\mathcal{R}_1}\!\!\overset{P_0}{\nleftarrow\!\!\!\!\!\!\nleftarrow} s \xrightarrow{\epsilon}_{\mathcal{R}_2} u_0\sigma = u$, and $t_0 \;{}_{\mathcal{R}}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow\!\rtimes\!\!\xrightarrow{\epsilon}_{\mathcal{R}} u_0$" but not $t_0 \leftrightarrow_{\mathcal{C}}^* u_0$. Take $t' = t_0\sigma$ and $u' = u_0\sigma$. Then $t_0\tau \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow t_0\sigma \;{}_{\mathcal{P}}\!\!\leftarrow s \rightarrow_{\mathcal{P}} u_0\sigma = u$ holds and by the assumption $t' \rightarrow_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow u'$ also holds. Hence (ii) follows.
   - Otherwise, whenever $P_0$, $t_0$, $u_0$, and $\sigma$ satisfy the conditions quoted in the last item, $t_0 \leftrightarrow_{\mathcal{C}}^* u_0$ holds. Because $\Gamma$ is not orthogonal, by Lemma 16(b) there exist $P_0$, $t_0$, $u_0$, $\sigma$, and $\tau$ such that $P_0 \subseteq P$, $t = t_0\tau \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow t_0\sigma \;{}_{\mathcal{R}_1}\!\!\overset{P_0}{\nleftarrow\!\!\!\!\!\!\nleftarrow} s \xrightarrow{\epsilon}_{\mathcal{R}_2} u_0\sigma = u$, $\sigma \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_1} \tau$. Thus $t_0 \leftrightarrow_{\mathcal{C}}^* u_0$ follows. Therefore, $t = t_0\tau \leftrightarrow_{\mathcal{C}}^* u_0\tau \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow u_0\sigma = u$, and hence (i) holds.
4. If $P = \{\epsilon\}$, $Q \nsubseteq \{\epsilon\}$, and $\Gamma$ is not orthogonal then the proof is analogous to the last case.
5. If $P \nsubseteq \{\epsilon\}$ and $Q \nsubseteq \{\epsilon\}$ then $s$, $t$, and $u$ can be written as $f(s_1, \ldots, s_n)$, $f(t_1, \ldots, t_n)$, and $f(u_1, \ldots, u_n)$ respectively, and $\Gamma_i \colon t_i \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow s_i \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_2} u_i$ holds for all $1 \leqslant i \leqslant n$. For every peak $\Gamma_i$ the induction hypothesis yields (i) or (ii). If (i) holds for all $\Gamma_i$ then (i) is concluded for $\Gamma$. Otherwise, some $\Gamma_i$ satisfies (ii). By taking $t' = f(s_1, \ldots, t_i, \ldots, s_n)$ and $u' = f(s_1, \ldots, u_i, \ldots, s_n)$ we have $t \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow t' \;{}_{\mathcal{P}}\!\!\leftarrow s \rightarrow_{\mathcal{P}} u' \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{P}} u$. From $t_i \rightarrow_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow u_i$ we obtain $t' \rightarrow_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow u'$. Hence $\Gamma$ satisfies (ii). ◀

The next theorem is a compositional confluence criterion based on parallel critical pair systems.

▶ **Theorem 38.** *Let $\mathcal{R}$ be a left-linear TRS and $\mathcal{C}$ a confluent TRS with $\mathcal{C} \subseteq \mathcal{R}$. The TRS $\mathcal{R}$ is confluent if ${}_{\mathcal{R}}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow\!\rtimes\!\!\xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\!\leftarrow$ and $\mathcal{P}/\mathcal{R}$ is terminating, where $\mathcal{P} = \mathsf{PCPS}(\mathcal{R}, \mathcal{C})$.*

**Proof.** Let $\bot$ be a fresh symbol and let $I = \mathcal{T}(\mathcal{F}, \mathcal{V}) \cup \{\bot\}$. We define the relation $>$ on $I$ as follows: $\alpha > \beta$ if $\alpha \neq \bot = \beta$ or $\alpha \rightarrow_{\mathcal{P}/\mathcal{R}}^+ \beta$. Since $\mathcal{P}/\mathcal{R}$ is terminating, $>$ is a well-founded order. Let $\mathcal{A} = (\mathcal{T}(\mathcal{F}, \mathcal{V}), \{\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\alpha\}_{\alpha \in I})$ be the ARS, where $\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\alpha$ is defined as follows: $s \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\alpha t$ if either $\alpha = \bot$ and $s \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{C}} t$, or $\alpha \neq \bot$ and $\alpha \rightarrow_{\mathcal{R}}^* s \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R} \setminus \mathcal{C}} t$. Since the commutation of $\mathcal{C}$ and $\mathcal{C}$ follows from confluence of $\mathcal{C}$, Lemma 5 yields the commutation of $\rightarrow_\bot$ and $\rightarrow_\bot$. According to Lemma 5 and Theorem 20, it is sufficient to show that every local peak

$$\Gamma \colon t \underset{\alpha}{\nleftarrow\!\!\!\!\!\!\nleftarrow} s \underset{\beta}{\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow} u$$

with $(\alpha, \beta) \in I^2 \setminus \{(\bot, \bot)\}$ is decreasing. By the definition of $\mathcal{A}$ we have $s \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_1} t$ and $s \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_2} u$ for some TRSs $\mathcal{R}_1, \mathcal{R}_2 \in \{\mathcal{R} \setminus \mathcal{C}, \mathcal{C}\}$. Using Lemma 37, we distinguish two cases.

1. Suppose that Lemma 37(i) holds for $\Gamma$. Then $t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_2} t' \leftrightarrow_{\mathcal{C}}^* u' \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow u$ holds for some $t'$ and $u'$. If $\mathcal{R}_2 = \mathcal{R} \setminus \mathcal{C}$ then $t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\beta t'$ follows from $\beta \rightarrow_{\mathcal{R}}^* s \rightarrow_{\mathcal{R}}^* t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R} \setminus \mathcal{C}} t'$. Otherwise, $\mathcal{R}_2 = \mathcal{C}$ yields $t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\bot t'$. In either case $t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\{\beta, \bot\}} t'$ is obtained. Similarly, $u \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\{\alpha, \bot\}} u'$ is obtained. Moreover, $t' \leftrightarrow\!\!\!\!\!\!\!\nleftrightarrow_\bot^* u'$ follows from $t' \leftrightarrow_{\mathcal{C}}^* u'$. Since $(\alpha, \beta) \neq (\bot, \bot)$ yields $\bot \in \curlyvee\alpha\beta$ and the reflexivity of $\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\bot$ yields $\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\{\delta, \bot\}} \subseteq \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\delta^= \cdot \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\bot$ for any $\delta$, we obtain the desirable conversion $t \overset{=}{\underset{\beta}{\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow}} t' \overset{*}{\underset{\curlyvee\alpha\beta}{\leftrightarrow\!\!\!\!\!\!\!\nleftrightarrow}} u' \overset{=}{\underset{\alpha}{\nleftarrow\!\!\!\!\!\!\nleftarrow}} u$. Hence, $\Gamma$ is decreasing.
2. Suppose that Lemma 37(ii) holds for $\Gamma$. We have $t \;{}_{\mathcal{R}_1}\!\!\nleftarrow\!\!\!\!\!\!\nleftarrow t' \;{}_{\mathcal{P}}\!\!\leftarrow s \rightarrow_{\mathcal{P}} u' \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\mathcal{R}_2} u$ and $t' \rightarrow_{\mathcal{R}}^* v \;{}_{\mathcal{R}}^*\!\!\leftarrow u'$ for some $t'$, $u'$, and $v$. As $(\alpha, \beta) \neq (\bot, \bot)$, we have $\alpha \rightarrow_{\mathcal{R}}^* s \rightarrow_{\mathcal{P}} t'$ or $\beta \rightarrow_{\mathcal{R}}^* s \rightarrow_{\mathcal{P}} t'$, from which $\alpha > t'$ or $\beta > t'$ follows. Thus, $t' \in \curlyvee\alpha\beta$. If $\mathcal{R}_2 = \mathcal{R} \setminus \mathcal{C}$ then $t' \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{t'} t$. Otherwise, $\mathcal{R}_2 = \mathcal{C}$ yields $t' \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\bot t$. So in either case $t \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\curlyvee\alpha\beta} t'$ holds. Consider terms $w$ and $w'$ with $t' \rightarrow_{\mathcal{R}}^* w \rightarrow_{\mathcal{R}} w' \rightarrow_{\mathcal{R}}^* v$. We have $w \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{t'} w'$ or $w \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_\bot w'$. So $w \twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow_{\curlyvee\alpha\beta} w'$ follows by $\{t', \bot\} \subseteq \curlyvee\alpha\beta$. Thus, $t \nleftarrow\!\!\!\!\!\!\nleftarrow_{\curlyvee\alpha\beta} t' \leftrightarrow\!\!\!\!\!\!\!\nleftrightarrow_{\curlyvee\alpha\beta}^* v$. In a similar way $u \nleftarrow\!\!\!\!\!\!\nleftarrow_{\curlyvee\alpha\beta} u' \leftrightarrow\!\!\!\!\!\!\!\nleftrightarrow_{\curlyvee\alpha\beta}^* v$ is obtained. Therefore $t \underset{\curlyvee\alpha\beta}{\nleftarrow\!\!\!\!\!\!\nleftarrow} t' \overset{*}{\underset{\curlyvee\alpha\beta}{\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow}} v \overset{*}{\underset{\curlyvee\alpha\beta}{\nleftarrow\!\!\!\!\!\!\nleftarrow}} u' \underset{\curlyvee\alpha\beta}{\twoheadrightarrow\!\!\!\!\!\!\!\nrightarrow} u$, and hence $\Gamma$ is decreasing. ◀

We claim that Theorem 34 is subsumed by Theorem 38. Suppose that $\mathcal{C}$ is the empty TRS. Trivially $\mathcal{C}$ is confluent. Because $\mathsf{PCPS}(\mathcal{R},\mathcal{C})$ is a subset of $\mathsf{PCPS}(\mathcal{R})$, termination of $\mathsf{PCPS}(\mathcal{R},\mathcal{C})/\mathcal{R}$ follows from that of $\mathsf{PCPS}(\mathcal{R})/\mathcal{R}$, which is equivalent to termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$. Finally, ${}_{\mathcal{R}}{\leftarrow\!\!\!+\!\!\!\rtimes}\xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow$ is a necessary condition of confluence. Thus, whenever Theorem 34 applies, Theorem 38 applies.

Theorem 38 also subsumes Theorem 24 too. Suppose that $\mathcal{C}$ is a confluent subsystem of $\mathcal{R}$. If ${}_{\mathcal{R}}{\leftarrow\!\!\!+\!\!\!\rtimes}\xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \leftrightarrow_{\mathcal{C}}^*$ then $\mathsf{PCPS}(\mathcal{R},\mathcal{C}) = \varnothing$, which leads to the termination of $\mathsf{PCPS}(\mathcal{R},\mathcal{C})/\mathcal{R}$. Hence, Theorem 38 applies. Note that if $\mathcal{C} = \mathcal{R}$ then $\mathsf{PCPS}(\mathcal{R},\mathcal{C}) = \varnothing$.

▶ **Example 39.** Consider the left-linear TRS $\mathcal{R}$:

$$\begin{array}{lll} 1\colon\ \mathsf{s}(\mathsf{p}(x)) \to x & 3\colon\ x + \mathsf{0} \to x & 5\colon\ x + \mathsf{s}(y) \to \mathsf{s}(x+y) \\ 2\colon\ \mathsf{p}(\mathsf{s}(x)) \to x & 4\colon\ \mathsf{0} + x \to x + \mathsf{0} & 6\colon\ x + \mathsf{p}(y) \to \mathsf{p}(x+y) \end{array}$$

We show the confluence of $\mathcal{R}$ by the combination of Theorem 38 and orthogonality. Let $\mathcal{C} = \{3\}$. The TRS $\mathsf{PCPS}(\mathcal{R},\mathcal{C})$ consists of the eight rules:

$$\begin{array}{ll} \mathsf{0} + \mathsf{s}(x) \to \mathsf{s}(\mathsf{0} + x) & x + \mathsf{s}(\mathsf{p}(y)) \to \mathsf{s}(x + \mathsf{p}(y)) \\ \mathsf{0} + \mathsf{s}(x) \to \mathsf{s}(x) + \mathsf{0} & x + \mathsf{s}(\mathsf{p}(y)) \to x + y \\ \mathsf{0} + \mathsf{p}(x) \to \mathsf{p}(\mathsf{0} + x) & x + \mathsf{p}(\mathsf{s}(y)) \to \mathsf{p}(x + \mathsf{s}(y)) \\ \mathsf{0} + \mathsf{p}(x) \to \mathsf{p}(x) + \mathsf{0} & x + \mathsf{p}(\mathsf{s}(y)) \to x + y \end{array}$$

Termination of $\mathsf{PCPS}(\mathcal{R},\mathcal{C})/\mathcal{R}$ can be shown by, e.g., the termination tool NaTT [38]. Since $\mathcal{C}$ is orthogonal and $\mathcal{R}$ is locally confluent, Theorem 38 applies. Note that the confluence of $\mathcal{R}$ can neither be shown by Theorem 28 nor Theorem 34. The former fails due to the lack of suitable labeling functions for the following diagrams:

$$\begin{array}{ccc} x + \mathsf{s}(\mathsf{p}(y)) & \xrightarrow[5]{\epsilon} & \mathsf{s}(x + \mathsf{p}(y)) \\ \{2\}\ \big\downarrow\!\!\!+\ 1 & & \big\downarrow\ 6 \\ x + y & \xleftarrow[1]{} & \mathsf{s}(\mathsf{p}(x + y)) \end{array} \qquad \begin{array}{ccc} x + \mathsf{p}(\mathsf{s}(y)) & \xrightarrow[6]{\epsilon} & \mathsf{p}(x + \mathsf{s}(y)) \\ \{2\}\ \big\downarrow\!\!\!+\ 2 & & \big\downarrow\ 5 \\ x + y & \xleftarrow[2]{} & \mathsf{p}(\mathsf{s}(x + y)) \end{array}$$

The latter fails due to non-termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$. The culprit is the rule $\mathsf{0} + \mathsf{0} \to \mathsf{0} + \mathsf{0}$ in $\mathsf{CPS}(\mathcal{R})$, originating from the critical peak $\mathsf{0} \leftarrow \mathsf{0} + \mathsf{0} \to \mathsf{0} + \mathsf{0}$. In contrast, the rule does not belong to $\mathsf{PCPS}(\mathcal{R},\mathcal{C})$ because the conversion $\mathsf{0} \leftrightarrow_{\mathcal{C}}^* \mathsf{0} + \mathsf{0}$ holds.

## 8 Experiments

In order to evaluate the presented approach we implemented the main three compositional confluence criteria (Theorems 24, 31, and 38) and their original versions (Theorems 22, 28, and 34) in our prototype confluence tool Hakusan.[1] The problem set used in experiments consists of 448 left-linear TRSs taken from the confluence problems database COPS [13]. Out of 448 TRSs, at least 179 are known to be non-confluent. The tests were run on a PC with Intel Core i7-1065G7 CPU (1.30 GHz) and 16 GB memory of RAM using timeouts of 120 seconds. Table 1 summarizes the results. The columns in the table stand for the following confluence criteria:

[1] The tool and the experimental data are available from: `https://www.jaist.ac.jp/project/saigawa/`

**Table 1** Experimental results on 448 left-linear TRSs.

| | **O** | **R** | **C** | **OO** | **RC** | **CR** | ACP | CoLL-Saigawa | CSI |
|---|---|---|---|---|---|---|---|---|---|
| # of proved TRSs | 20 | 132 | 58 | 85 | 149 | 140 | 195 | 168 | 209 |
| timeouts | 0 | 20 | 8 | 13 | 82 | 32 | 47 | 169 | 3 |

- **O**: Orthogonality (Theorem 22).
- **R**: Rule labeling (Theorem 28).
- **C**: The criterion by critical pair systems (Theorem 34).
- **OO**: Successive application of Theorem 24, as illustrated in Example 25.
- **RC**: Theorem 31, where confluence of a subsystem $\mathcal{C}$ is shown by Theorem 38 with the empty subsystem.
- **CR**: Theorem 38, where confluence of a subsystem $\mathcal{C}$ is shown by Theorem 31 with the empty subsystem.

For the sake of comparison the results of the confluence tools ACP version 0.62 [3], CoLL-Saigawa version 1.6 [27], and CSI version 1.25 [39] are also included in the table.

We briefly explain how these criteria are automated in our tool. Suitable subsystems for the compositional criteria are searched by enumeration. Relative termination, required by Theorems 34 and 38, is checked by employing the termination tool NaTT version 1.9 [38]. Joinability of each (parallel) critical pairs $(t, u)$ is tested by the relation:

$$t \xrightarrow{\leqslant 5} \cdot \xleftarrow{\leqslant 5} u$$

For rule labeling, the decreasingness of each parallel critical peak $t \;{}_{\phi,k}\overset{P}{\leftarrow\!\!\!\!+\!\!\!\!+} s \xrightarrow{\epsilon}_{\psi,m} u$ is checked by existence of a conversion of the form

$$t \xrightarrow[\curlyvee k]{}^{i_1} \cdot \underset{\psi,m}{\dashrightarrow\!\!\!\!\!\!+\!\!\!+}{}^{i_2} \cdot \xrightarrow[\curlyvee km]{}^{i_3} \cdot {}^{j_3}\xleftarrow[\curlyvee km]{} v \;{}^{j_2}\underset{\phi,k}{\overset{P'}{\leftarrow\!\!\!\!+\!\!\!\!+}} \cdot {}^{j_1}\xleftarrow[\curlyvee m]{} u$$

such that $i_1, i_3, j_1, j_3 \in \mathbb{N}$, $i_2, j_2 \in \{0, 1\}$, $i_1 + i_2 + i_3 \leqslant 5$, $j_1 + j_2 + j_3 \leqslant 5$, and the inclusion $\mathcal{V}\mathsf{ar}(v, P') \subseteq \mathcal{V}\mathsf{ar}(s, P)$ holds. This is encoded into linear arithmetic constraints [11], and they are solved by the SMT solver Z3 version 4.8.11 [5].

As theoretically expected, in the experiments **O**, **R**, and **C** are subsumed by their compositional versions **OO**, **RC**, and **CR**, respectively. Moreover, **OO** is subsumed by **R**, **RC**, and **CR**. Due to timeouts, **CR** misses three systems of which **R** can prove confluence. While the union of **R** and **C** amounts to 142, the union of **RC** and **CR** amounts to 150. Differences between **RC** and **CR** are summarized as follows:

- Three systems are proved by **RC** but not by **CR** nor **R**.[2] One of them is the next TRS (COPS number 994). **RC** uses the subsystem $\{2, 4, 6\}$ whose confluence is shown by **C**.

$$\begin{array}{lll} 1\colon\ \mathsf{a}(\mathsf{b}(x)) \to \mathsf{a}(\mathsf{c}(x)) & 3\colon\ \mathsf{c}(\mathsf{b}(x)) \to \mathsf{a}(\mathsf{b}(x)) & 5\colon\ \mathsf{c}(\mathsf{c}(x)) \to \mathsf{c}(\mathsf{c}(x)) \\ 2\colon\ \mathsf{a}(\mathsf{c}(x)) \to \mathsf{c}(\mathsf{b}(x)) & 4\colon\ \mathsf{b}(\mathsf{c}(x)) \to \mathsf{a}(\mathsf{c}(x)) & 6\colon\ \mathsf{c}(\mathsf{c}(x)) \to \mathsf{c}(\mathsf{b}(x)) \end{array}$$

- The only TRS where **CR** is advantageous to **RC** is COPS number 132:

$$\begin{array}{ll} 1\colon\ -(x + y) \to (-x) + (-y) & 3\colon\ -(-x) \to x \\ 2\colon\ (x + y) + z \to x + (y + z) & 4\colon\ x + y \to y + x \end{array}$$

Its confluence is shown by the composition of Theorem 38 and Theorem 28, the latter of which proves the subsystem $\{1, 2, 4\}$ confluent.

[2] The three systems are COPS numbers 994, 1001, and 1029. The aforementioned confluence tools also fail to prove confluence of these systems.

## 9 Conclusion

We studied how compositional confluence criteria can be derived from confluence criteria based on the decreasing diagrams technique, and showed that Toyama's almost parallel closedness theorem is subsumed by his earlier theorem based on parallel critical pairs. We conclude the paper by mentioning related work and future work.

**Simultaneous critical pairs.** van Oostrom [36] showed the almost development closedness theorem: A left-linear TRS is confluent if the inclusions

$$\xleftarrow{\epsilon}\rtimes\xrightarrow{\epsilon} \subseteq \xrightarrow{*} \cdot \mathrel{\leftarrow\!\!\circ\!\!-} \qquad\qquad \xleftarrow{>\epsilon}\rtimes\xrightarrow{\epsilon} \subseteq \mathrel{-\!\!\circ\!\!\rightarrow}$$

hold, where $\mathrel{-\!\!\circ\!\!\rightarrow}$ stands for the multi-step [29, Section 4.7.2]. Okui [23] showed the simultaneous closedness theorem: A left-linear TRS is confluent if the inclusion

$$\mathrel{\leftarrow\!\!\circ\!\!-}\rtimes\!\rightarrow \subseteq \xrightarrow{*} \cdot \mathrel{\leftarrow\!\!\circ\!\!-}$$

holds, where $\mathrel{\leftarrow\!\!\circ\!\!-}\rtimes\!\rightarrow$ stands for the set of simultaneous critical pairs [23]. As this inclusion characterizes the inclusion $\mathrel{\leftarrow\!\!\circ\!\!-} \cdot \rightarrow \subseteq \rightarrow^* \cdot \mathrel{\leftarrow\!\!\circ\!\!-}$, simultaneous closedness subsumes almost development closedness. The main result in Section 3 is considered as a counterpart of this relationship in the setting of parallel critical pairs.

**Critical-pair-closing systems.** A TRS $\mathcal{C}$ is called *critical-pair-closing* for a TRS $\mathcal{R}$ if

$${}_{\mathcal{R}}\!\leftarrow\rtimes\xrightarrow{\epsilon}_{\mathcal{R}} \subseteq \leftrightarrow^*_{\mathcal{C}}$$

holds. It is known that a left-linear TRS $\mathcal{R}$ is confluent if $\mathcal{C}_{\mathsf{d}}/\mathcal{R}$ is terminating for some confluent critical-pair-closing TRS $\mathcal{C}$ with $\mathcal{C} \subseteq \mathcal{R}$, see [14]. Here $\mathcal{C}_{\mathsf{d}}$ denotes the set of all duplicating rules in $\mathcal{C}$. Theorem 24 imposes closedness by $\mathcal{C}$ on all *parallel* critical pairs in return to removal of the relative termination condition. Investigating whether the latter subsumes the former is our future work.

**Rule labeling.** Dowek et al. [7, Theorem 38] extended rule labeling based on parallel critical pairs [40] to take higher-order rewrite systems. If we restrict their method to a first-order setting, it corresponds to the case that a complete TRS is employed for $\mathcal{C}$ in Theorem 31, and thus, it can be seen as a generalization of Corollary 26 by Toyama [33].

**Critical pair systems.** The second author and Middeldorp [12] generalized Theorem 34 by replacing $\mathsf{CPS}(\mathcal{R})$ by the following subset:

$$\mathsf{CPS}'(\mathcal{R}) = \{s \to t, s \to u \mid t \;{}_{\mathcal{R}}\!\leftarrow s \xrightarrow{\epsilon}_{\mathcal{R}} u \text{ is a critical peak but not } t \mathrel{-\!\!\circ\!\!\rightarrow}_{\mathcal{R}} u\}$$

This variant subsumes van Oostrom's development closedness theorem [36]. We anticipate that in a similar way our compositional variant (Theorem 38) is extended to subsume the parallel closedness theorem based on parallel critical pairs (Theorem 14).

**Modularity and Automation.** Compositional criteria are conceived as a criterion that confluence of a subsystem implies confluence of the original system. In their automation searching suitable subsystems is a serious bottleneck. If a criterion for the converse direction is established, the bottleneck is resolved as a confluence problem reduces to that of a subsystem. Modularity-based decomposition methods [31, 34, 1, 22, 8] are capable of this type of reduction. Integrating modularity results in compositional criteria is our another future work.

## References


1 T. Aoto and Y. Toyama. Persistency of confluence. *Journal of Universal Computer Science*, 3(11):1134–1147, 1997.

2 T. Aoto and Y. Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8, 2012.

3 T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *LNCS*, pages 93–102, 2009.

4 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

5 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340, 2008. The website of Z3 is: `https://github.com/Z3Prover/z3`.

6 N. Dershowitz. Open. Closed. Open. In *Proc. 16th International Conference on Rewriting Techniques and Applications*, volume 3467 of *LNCS*, pages 276–393, 2005.

7 G. Dowek, G. Férey, J.-P. Jouannaud, and J. Liu. Confluence of left-linear higher-order rewrite theories by checking their nested critical pairs. *Mathematical Structures in Computer Science*, 2022.

8 B. Felgenhauer, A. Middeldorp, H. Zankl, and V. van Oostrom. Layer systems for proving confluence. *ACM Trans. Comput. Logic*, 16(2):1–32, 2015.

9 B. Felgenhauer and V. van Oostrom. Proof orders for decreasing diagrams. In *Proc. 24th International Conference on Rewriting Techniques and Applications*, volume 21 of *LIPIcs*, pages 174–189, 2013.

10 B. Gramlich. Confluence without termination via parallel critical pairs. In *Proc. 21st International Colloquium on Trees in Algebra and Programming*, volume 1059 of *LNCS*, pages 211–225, 1996.

11 N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47:481–501, 2011.

12 N. Hirokawa and A. Middeldorp. Commutation via relative termination. In *Proc. 2nd International Workshop on Confluence*, pages 29–34, 2013.

13 N. Hirokawa, J. Nagele, and A. Middeldorp. Cops and CoCoWeb: Infrastructure for confluence tools. In *Proc. 9th International Joint Conference on Automated Reasoning*, volume 10900 of *LNCS (LNAI)*, pages 346–353, 2018. The website of COPS is: `https://cops.uibk.ac.at/`.

14 N. Hirokawa, J. Nagele, V. van Oostrom, and M. Oyamaguchi. Confluence by critical pair analysis revisited. In *Proc. 27th International Conference on Automated Deduction*, volume 11716 of *LNCS*, pages 319–336, 2019.

15 G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

16 J.-P. Jouannaud and J. Liu. From diagrammatic confluence to modularity. *Theoretical Computer Science*, 464:20–34, 2012.

17 S. Kahrs. Confluence of curried term-rewriting systems. *Journal of Symbolic Computation*, 19:601–623, 1995.

18 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.

19 J. Liu and J.-P. Jouannaud. Confluence: The unifying, expressive power of locality. In *Specification, Algebra, and Software*, volume 8375 of *LNCS*, pages 337–358, 2014.

20 J. Nagele, B. Felgenhauer, and A. Middeldorp. Improving automatic confluence analysis of rewrite systems by redundant rules. In *Proc. 26th International Conference on Rewriting Techniques and Applications*, volume 36 of *LIPIcs*, pages 257–268, 2015.

21 M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

22 E. Ohlebusch. *Advanced Topics in Term Rewriting.* Springer, 2002.

**23** S. Okui. Simultaneous critical pairs and Church–Rosser property. In *Proc. 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 2–16, 1998. doi:10.1007/BFb0052357.

**24** M. Oyamaguchi and Y. Ohta. A new parallel closed condition for Church-Rosser of left-linear term rewriting systems. In *Proc. 10th International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 187–201, 1997.

**25** M. Oyamaguchi and Y. Ohta. On the Church-Rosser property of left-linear term rewriting systems. *IEICE Transactions on Information and Systems*, E86-D(1):131–135, 2003.

**26** B. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, pages 160–187, 1973.

**27** K. Shintani and N. Hirokawa. CoLL: A confluence tool for left-linear term rewrite systems. In *Proc. 25th International Conference on Automated Deduction*, volume 9195 of *LNCS (LNAI)*, pages 127–136, 2015.

**28** M. Takahashi. λ-calculi with conditional rules. In *Proc. International Conference on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 406–417, 1993.

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

**30** Y. Toyama. On the Church–Rosser property of term rewriting systems. In *NTT ECL Technical Report*, volume No. 17672. NTT, 1981. Japanese.

**31** Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, 1987.

**32** Y. Toyama. Commutativity of term rewriting systems. In *Programming of Future Generation Computers II*, pages 393–407. North-Holland, 1988.

**33** Y. Toyama. Confluence criteria based on parallel critical pair closing, March 2017. Presented at the 46th TRS Meeting: https://www.trs.cm.is.nagoya-u.ac.jp/event/46thTRSmeeting/.

**34** J. van de Pol. Modularity in many-sorted term rewriting systems. Master's thesis, Utrecht University, 1992.

**35** V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.

**36** V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

**37** V. van Oostrom. Confluence by decreasing diagrams, converted. In *Proc. 19th International Conference on Rewriting Techniques and Applications*, volume 5117 of *LNCS*, pages 306–320, 2008.

**38** A. Yamada, K. Kusakari, and T.Sakabe. Nagoya termination tool. In *Proc. 25th International Conference on Rewriting Techniques and Applications*, volume 8560 of *LNCS*, pages 446–475, 2014. The website of NaTT is: https://www.trs.cm.is.nagoya-u.ac.jp/NaTT/.

**39** H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI - a confluence tool. In *Proc. 23th International Conference on Automated Deduction*, volume 6803 of *LNCS (LNAI)*, pages 499–505, 2011.

**40** H. Zankl, B. Felgenhauer, and A. Middeldorp. Labelings for decreasing diagrams. *Journal of Automated Reasoning*, 54(2):101–133, 2015.

# Rewriting for Monoidal Closed Categories

**Mario Alvarez-Picallo**
Programming Languages Laboratory, Huawei Research Centre, UK

**Dan Ghica**
Department of Computer Science, University of Birmingham, UK
Programming Languages Laboratory, Huawei Research Centre, Reading, UK

**David Sprunger**
Department of Computer Science, University of Birmingham, UK

**Fabio Zanasi**
Department of Computer Science, University College London, UK

## Abstract

This paper develops a formal string diagram language for monoidal closed categories. Previous work has shown that string diagrams for freely generated symmetric monoidal categories can be viewed as hypergraphs with interfaces, and the axioms of these categories can be realized by rewriting systems. This work proposes hierarchical hypergraphs as a suitable formalization of string diagrams for monoidal closed categories. We then show double pushout rewriting captures the axioms of these closed categories.

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

**Keywords and phrases** string diagrams, rewriting, hierarchical hypergraph, monoidal closed category

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.29

**Funding** This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/V001612/1 and EP/V002376/1].

## 1 Introduction

Symmetric monoidal categories are algebraic settings for abstract functions (called *morphisms* or *arrows*) between different sources and targets (called *objects*), which support both sequential composition and parallel composition. These categories have two widespread notations: terms and string diagrams. Term notation is generally regarded as the *de facto* standard; string diagrams were once thought of as a "private device" useful for quick calculation, but which "should be provided with a firm theoretical foundation" in order to be credibly used in public [23].

Since (and including) Joyal and Street's landmark paper in 1991, much scholarly work has gone into the formalization of string diagrams. This has alternately supported and been motivated by a proliferation of applications in compositional system modelling, for example in the representation of Petri nets [28], (analog) electrical circuits [4], digital circuits [17], quantum processes [14], differentiable programs [36], and signal flow graphs [8].

Separately, monoidal closed categories, particularly cartesian closed categories, have been used in theoretical computer science as models for the simply typed lambda calculus and functional programming languages. String diagrams in these contexts give an alternative method for specifying, implementing, and reasoning about complex program transformations, such as automatic differentiation [2]. There is unfilled need for a diagrammatic language to reason in closed categories for these applications.

We propose *hierarchical string diagrams* as a language to represent morphisms in these closed categories, which extend ordinary monoidal string diagrams with a bubble operation to represent curried terms. Following [7, 5, 6], which formalizes string diagrams as hypergraphs

© Mario Alvarez-Picallo, Dan Ghica, David Sprunger, and Fabio Zanasi;
licensed under Creative Commons License CC-BY 4.0

and gives a formal notion of hypergraph rewriting, we formalize these hierarchical string diagrams with *hierarchical hypergraphs*. Finally, we present a double-pushout rewriting system which is sound and complete for the axioms of monoidal closed categories, and also extensible to any other equational system.

**Outline.** In Section 2, we describe monoidal (closed) categories and their string diagrams. We then formalize string diagrams for monoidal categories in Section 3 as certain hypergraphs. We introduce hierarchical hypergraphs and formalize their rewriting in Section 4. Finally, we establish connections to term rewriting. Further comparisons with related work can be found in Section 6.

# 2 Monoidal closed categories and their string diagrams

In this section, we recall (strict symmetric) monoidal categories and their basic string diagrams. Then we recall monoidal closed categories and reintroduce a proposed graphical syntax for their morphisms, *hierarchical string diagrams*.

We assume familiarity with basic category theory. In this paper, the collection of objects of a category $\mathcal{C}$ is denoted $|\mathcal{C}|$ and the set of morphisms from $A$ to $B$ is $\mathcal{C}(A, B)$. The sequential composition of $f : A \to B$ and $g : B \to C$ is $f; g : A \to C$.[1] The identity morphism on an object $A$ is denoted $1_A$. The set of lists with entries in the set $X$ is denoted $X^*$ and similarly the application of a function $f : X \to Y$ to a list from $X^*$ is denoted $f^*$.

## 2.1 Monoidal categories

▶ **Definition 1.** *A category $\mathcal{C}$ is* monoidal *means it is equipped with both a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ which is associative (up to a natural isomorphism) and an object $I$ which is both a left and right unit for $\otimes$ (up to a natural isomorphism).*

*The category is called* strict monoidal *when these natural isomorphisms are identities.*

*A strict monoidal category is called* symmetric *if there is a natural isomorphism $\sigma_{A,B} : A \otimes B \to B \otimes A$ satisfying three properties: (1) $1_A = \sigma_{A,I}$, (2) $\sigma_{A,B}; \sigma_{B,A} = 1_{A\otimes B}$ (3) $(1_A \otimes \sigma_{B,C}); (\sigma_{A,C} \otimes 1_B) = \sigma_{A\otimes B,C}$.*

To reduce the use of grouping symbols, we adopt the convention that $\otimes$ binds tighter than ; meaning, e.g., $f \otimes g; h$ is $(f \otimes g); h$, not $f \otimes (g; h)$. We will refer to $f \otimes g$ as the *parallel composition (of $f$ and $g$)*, and we call $f; g$ the *sequential composition (of $f$ and $g$)*.

For simplicity, we restrict our attention to strict monoidal categories, noting that every monoidal category is monoidally equivalent to a strict monoidal category [26]. In every strict monoidal category, the equations in the first two rows of Figure 1 hold; the last row holds for strict symmetric monoidal categories.

$$
\begin{array}{lll}
(f; g); h = f; (g; h) & 1_A; f = f = f; 1_B & (f; g) \otimes (h; k) = (f \otimes h); (g \otimes k) \\
(f \otimes g) \otimes h = f \otimes (g \otimes h) & 1_I \otimes f = f = f \otimes 1_I & 1_{A\otimes B} = \sigma_{A,B}; \sigma_{B,A} \\
f \otimes g; \sigma_{C,D} = \sigma_{A,B}; g \otimes f & 1_{A\otimes B} = 1_A \otimes 1_B & \sigma_{A,B\otimes C} = \sigma_{A,B} \otimes 1_C; 1_B \otimes \sigma_{A,C}
\end{array}
$$

**Figure 1** Laws of strict (symmetric) monoidal categories.

[1] In this paper, we use the relational order of composition, as opposed to the more common notation $g \circ f$.

String diagrams are a graphical syntax for morphisms in strict monoidal categories. In this syntax, morphisms are represented by labelled boxes with labelled wires on either side: the morphism $f : A \to B$ is denoted . A sequential composition is formed by vertically stacking the component string diagrams and joining corresponding wires, as in . Parallel composition places the diagrams side-by-side as in .

Some advantages of string diagram notation become clear when considering the equations of Figure 1. With the common conventions that the identity morphism on an object can be denoted by a plain wire and the identity morphism on the tensor unit $I$ can be denoted by a blank space, the string diagrams representing either side of many of these equations appear the same up to some topological deformations.

A challenge in the use of string diagrams is the *wire matching problem*, where the operation of "stack and join corresponding wires" may not be well-defined. For example, if we represent $k : X \to A \otimes C$ by , we cannot match wires to compose with . A common way to avoid this problem is to restrict attention to freely generated monoidal categories where objects have unique representations. This ensures that every string diagram has a canonical arrangement of input and output wires, and therefore ensures that string diagrams for sequentially composable morphisms have matchable inputs and outputs.

Strict symmetric monoidal categories (SMCs) stipulate the existence of a family of morphisms: $\sigma_{A,B}$. Following common convention, instead of using a box labelled $\sigma_{A,B}$ as a string diagram for this morphism, we use a pair of crossing wires: .

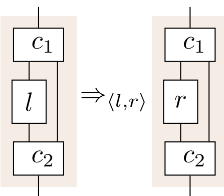



**Figure 2** A rewriting step.

Similar to [7, 5, 6], we consider *string diagram rewriting*. In a symmetric monoidal category, a *string diagram rewriting rule* is a pair of parallel morphisms $\langle l, r\rangle$. A morphism $d$ *rewrites directly* to another morphism $e$ under such a rule (denoted $d \Rightarrow_{\langle l,r\rangle} e$) if there are morphisms $c_1, c_2$ and an object $k$ such that $d = c_1; 1_k \otimes l; c_2$ and $e = c_1; 1_k \otimes r; c_2$. A *string diagram rewriting system* is a collection of rewriting rules; a morphism $d$ *rewrites* to a parallel morphism $e$ in a rewriting system if there is a sequence of direct rewrites in the system starting with $d$ and ending in $e$.

## 2.2 Term rewriting for morphisms in symmetric monoidal categories

Commonly, morphisms in a category are described with terms. (For example, the equations of Figure 1.) Reasoning with these terms is performed by rewriting them with known equations. We recall a formal treatment of this next.

A *monoidal signature* $\Sigma = (\Sigma_O, \Sigma_M, t)$ consists of a set of types $\Sigma_O$, a set of constant symbols $\Sigma_M$ and an assignment $t : \Sigma_M \to \Sigma_O^* \times \Sigma_O^*$ giving a list of input and output types to each constant symbol; abusing notation we denote this assignment by $\xi :_t A_1 \otimes \cdots \otimes A_n \to B_1 \otimes \cdots \otimes B_m$ where $\xi \in \Sigma_M$ and $A_i, B_j \in \Sigma_O$. We will often drop the subscript $t$. Again abusing notation, we denote the empty list of types by $I$.

*Morphism terms* $\tau$ in this signature are generated by the following BNF grammar: $\tau ::= \xi \mid f \mid 1_I \mid 1_A \mid \sigma_{A,B} \mid \tau_1;\tau_2 \mid \tau_1 \otimes \tau_2$ where $\xi \in \Sigma_M$ ranges over constants in the signature, $f$ ranges over a set of variables, and $A, B \in \Sigma_O$ range over types in the signature. Terms are given lists of input and output types from $\Sigma_O$; given a context $\Gamma$ (i.e. a typing of the variables denoted $f :_\Gamma X \to Y$), a term is said to be *well-typed* as usual. A term is a *ground term* if it contains no variables. An *equation* is a pair of terms of the same type in the same context. Substitution of a term ($\beta$) for a variable ($f$) in another term ($\alpha$) is defined as usual and denoted $\alpha[\beta/f]$.

The free strict symmetric monoidal category $S_\Sigma$ on a signature $\Sigma$ has as objects lists of types from the signature and as morphisms equivalence classes of well-typed ground terms in the signature. The equivalence classes are the congruence classes generated by the equations of Figure 1, except the last two equations in the last column. Note the set of terms does not include identities or symmetries at product types ($1_{A\otimes B}$ or $\sigma_{A,B\otimes C}$): these are defined as composites of other generators via the last two equations.

A *term rewrite rule* is a pair of morphism terms $\langle \lambda, \rho \rangle$. A *substitution instance* of a rewrite rule $\langle \lambda, \rho \rangle$ is a pair of morphism terms $\langle \zeta, \xi \rangle$ such that applying the same substitutions to $\lambda$ and $\rho$ yields $\zeta$ and $\xi$, respectively. A *term rewrite system* $R$ is a set of rewrite rules. A *context* is a term with a single occurrence of a designated variable $\bullet$ called its *hole*. We say a term $\alpha$ *rewrites directly* to the term $\beta$ in the system $R$ if there is a context $C$ and a substitution instance of a rewrite rule $\langle \lambda, \rho \rangle \in R$ such that $C[\zeta/\bullet] = \alpha$ and $C[\xi/\bullet] = \beta$. In this case, we write $\alpha \to_R \beta$. A term $\alpha$ *rewrites* to the term $\beta$ in a system $R$ if there is a sequence of direct rewrites starting from $\alpha$ and ending in $\beta$.

If $\alpha$ and $\beta$ are morphism terms that rewrite to one another via the bidirectional rewrite system formed from the equations of Figure 1), we say they are *equivalent modulo the laws of SMCs* and write $\alpha =_{SMC} \beta$.

Though many mathematicians default to reasoning with terms, string diagram rewriting possesses several powerful advantages. Emulating a string diagram rewrite step with terms may require several intermediate steps invoking laws of SMCs to find the appropriate representative of the term's equivalence class. This redex search is made drastically easier in string diagrams since morphisms are already quotiented by SMC laws.

However, string diagrams have disadvantages as well. Morphisms from a category are not a natural candidate for automated manipulation, whereas terms have a clear inductive structure. Additionally, due to the lack of variables, universally quantified equations (such as $f; !_B = !_A$) are emulated in string diagram rewriting system with a collection of rewrite rules, possibly even a rule schema, rather than with a single rewrite rule.

Implementing string diagrams with hypergraphs [7, 5, 6] ameliorates many of these computational disadvantages while retaining the automatic enforcement of SMC laws. We show how these hypergraphs can be defined inductively on morphism terms, while satisfying the SMC laws and thus serving as a functorial semantics for string diagrams. Following [7, 5, 6], we then use *double-pushout (DPO) graph rewriting* to rewrite hypergraphs in analogy with term rewriting and string diagram rewriting.

## 2.3 Monoidal closed categories

▶ **Definition 2.** *A* monoidal (right) closed category *is a monoidal category* $\mathcal{C}$ *satisfying for every pair of objects* $B, C$ *there is an object* $B \Rightarrow C$ *and a morphism* $ev_{B,C} : (B \Rightarrow C) \otimes B \to C$*, and for every triple of objects* $A, B, C$ *there is an operation* $\Lambda_{A,B,C} : \mathcal{C}(A \otimes B, C) \to \mathcal{C}(A, B \Rightarrow C)$ *satisfying three equations for all* $f : A \otimes B \to C$ *and* $g : Z \to A$*:*

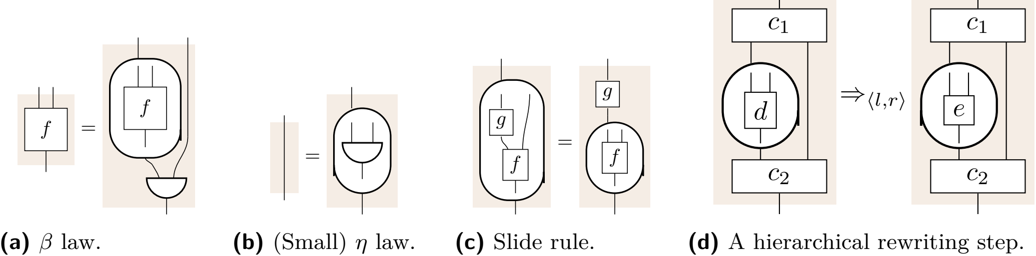
1. $f = \Lambda_{A,B,C}(f) \otimes 1_B; ev_{B,C}$
2. $1_{B \Rightarrow C} = \Lambda_{B \Rightarrow C,B,C}(ev_{B,C})$
3. $\Lambda_{Z,B,C}(g \otimes 1_B; f) = g; \Lambda_{A,B,C}(f)$

A more common equivalent definition is that the right tensor functor $- \otimes B : \mathcal{C} \to \mathcal{C}$ has a right adjoint $B \Rightarrow -$. The currying operation $\Lambda$ represents one direction of the homset presentation of this adjunction. The other direction is $h \mapsto h \otimes 1_B; ev_{B,C}$.

If tensoring on the left has a right adjoint, the category is left closed. A symmetric monoidal category is right closed if and only if it left closed; we simply call it *closed*.

This presentation of monoidal closed categories induces a manageable set of changes to both the term calculus and the string diagram calculus. Though it is possible to present adjunctions via string diagrams, for example with functorial boxes [27], here we only need a representation for the currying operation $\Lambda$ and some extra objects and morphisms.

Morphism terms for monoidal categories can be extended to terms for monoidal *closed* categories by adding a type-forming operation ($\Rightarrow$), a new collection of constant terms ($ev$), and a new term-forming operation ($\Lambda$). The BNF generating *closed morphism terms* is then $\rho ::= \xi \mid f \mid 1_I \mid 1_A \mid \sigma_{A,B} \mid ev_{A,B} \mid \rho_1; \rho_2 \mid \rho_1 \otimes \rho_2 \mid \Lambda(\rho)$. The free symmetric closed monoidal category can be defined again as equivalence classes of well-typed ground closed morphism terms; term rewriting is again similar to the symmetric monoidal case.




**(a)** $\beta$ law. **(b)** (Small) $\eta$ law. **(c)** Slide rule. **(d)** A hierarchical rewriting step.

**Figure 3** Diagrammatic equations for monoidal closed categories and hierarchical rewriting.

For string diagrams, the new objects mean wires have some new labels available. For convenience, we introduce the new morphism box shape , as syntactic sugar for a box labelled $ev$. We diagrammatically represent $\Lambda$ with a bubble operation. That is, if $f : A \otimes B \to C$, we write for $\Lambda(f)$. We call these *hierarchical string diagrams* since string diagrams may appear within wires of other string diagrams. The three equations of Definition 2 are interpreted diagrammatically as in Figure 3a-3c.

Rewriting of these hierarchical string diagrams is similarly an extension of rewriting string diagrams. A hierarchical rewriting rule is a pair of hierarchical string diagrams $\langle l, r \rangle$. A hierarchical string diagram directly rewrites to another if it can be put in the form of Figure 2 or the form of Figure 3d where $d$ is a hierarchical string diagram that directly rewrites to $e$ with the same rule.

## 3 String diagrams as hypergraphs

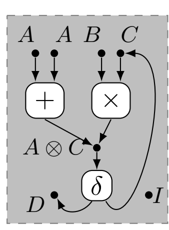
In this section, we formalize string diagrams as *monogamous directed acyclic hypergraphs* (Definition 7). We further define a rewriting scheme for these hypergraphs based on double pushout rewriting. In the next section, we will extend both of these notions to treat hierarchical string diagrams as hierarchical hypergraphs with a similar rewriting scheme.

### 3.1 Hypergraphs with interfaces

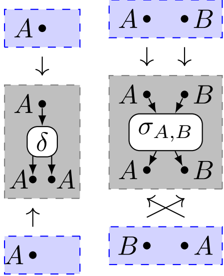
▶ **Definition 3.** *A* (directed) hypergraph (with labels from $\Sigma = (\Sigma_V, \Sigma_E)$) *is a tuple* $(V, E, s, t, \ell_V, \ell_E)$ *consisting of finite sets of* vertices $V$ *and* edges $E$*,* source *and* target functions $s, t : E \to V^*$ *and* vertex *and* edge labelling functions $\ell_V : V \to \Sigma_V$ *and* $\ell_E : E \to \Sigma_E$*.*

When considering multiple hypergraphs, we distinguish their components by subscripting, so $V_{\mathcal{G}}$ is the vertices of $\mathcal{G}$, $s_{\mathcal{H}}$ is the source function for $\mathcal{H}$, etc. Note that our hypergraphs' edges come with *lists* (rather than sets) of source and target vertices. Despite this, we write $v \in s(e)$ when the vertex $v$ occurs in the list of source vertices of the edge $e$.

We will depict hypergraphs graphically as in Figure 4a. In this hypergraph, there are three edges (white boxes labelled $+$, $\times$ and $\delta$), and seven vertices (black dots). The black arrows indicate the source/target relationship: a vertex is a source of an edge if there is an arrow from the dot to the box and a target if the arrow goes from the box to the dot. The leftmost arrow at the top of the box corresponds to the first source vertex in the list, and similarly the leftmost bottom arrow is the first target.




**(a)** Hypergraph without interfaces.




**(b)** Hypergraphs with interfaces.

**Figure 4** Example hypergraphs.

We borrow terminology from graph theory for hypergraphs. We say a hypergraph is *discrete* if the edge set is empty. A *directed path* in a hypergraph is a finite list of alternating vertices and edges $(v_0, e_0, v_1, \ldots, v_k)$ with the property that $v_i \in s(e_i)$ and $v_{i+1} \in t(e_i)$ for all $0 \leq i < k$. The *length* of the path is the number of edges in the path. A hypergraph is *directed acyclic* if there are no postitive-length directed paths from a vertex to itself.

A *sub-hypergraph* $\mathcal{G}$ of the hypergraph $\mathcal{H}$ is a subset of $\mathcal{H}$'s vertices and edges such that the restrictions of $s$ and $t$ to $\mathcal{G}$ make it a hypergraph. A sub-hypergraph $\mathcal{G}$ is *convex (in $\mathcal{H}$)* if all directed paths in $\mathcal{H}$ between vertices in $\mathcal{G}$ are directed paths in $\mathcal{G}$.

The *in-degree* of a vertex $v$ is the number of pairs $(e, i) \in E \times \mathbb{N}$ such that $v$ is entry $i$ in the list $t(e)$. Similarly, the *out-degree* is the number of occurrences of $v$ in source lists.

▶ **Definition 4.** *Suppose $\mathcal{F}$ and $\mathcal{G}$ are two hypergraphs with labels from $\Sigma$. A* hypergraph morphism *from $\mathcal{F}$ to $\mathcal{G}$ is a pair of functions $\phi_V : V_{\mathcal{F}} \to V_{\mathcal{G}}$ and $\phi_E : E_{\mathcal{F}} \to E_{\mathcal{G}}$ compatible with source and target labelling in the sense that (1) $s_{\mathcal{F}}; \phi_V^* = \phi_E; s_{\mathcal{G}}$, (2) $t_{\mathcal{F}}; \phi_V^* = \phi_E; t_{\mathcal{G}}$ (3) $\ell_{V,\mathcal{F}} = \phi_V; \ell_{V,\mathcal{G}}$, and (4) $\ell_{E,\mathcal{F}} = \phi_E; \ell_{E,\mathcal{G}}$*

Recall $\phi_V^* : V_\mathcal{F}^* \to V_\mathcal{G}^*$ is the elementwise application of $\phi_V$ to the argument list.

Hypergraphs with a fixed set of labels and the morphisms between them form a category which we denote by $\mathsf{Hyp}_\Sigma$.

▶ **Definition 5.** *A* hypergraph with (ordered) interfaces *is a cospan* $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ *in* $\mathsf{Hyp}_\Sigma$ *where the* $n$ *and* $m$ *are discrete hypergraphs with a specified total ordering on their vertices, and* $f$ *and* $g$ *are monos.*

We often refer to the hypergraphs $n$ and $m$ in the cospan as the *interfaces*, and the hypergraph $\mathcal{F}$ as just the hypergraph. Graphically, we distinguish interfaces with a blue background, as in Figure 4b. Since the interfaces are finite sets with a total order, we may also think of their vertex set as a list. (This happens, for example, in Definition 6.)

## 3.2 Monogamous directed acyclic hypergraphs

Next we will consider hypergraphs with labels drawn from a signature. When these hypergraphs have certain extra properties, we can think of them as syntax for representing morphisms from the free SMC on that signature. First, some preliminary notions.

▶ **Definition 6.** *Suppose* $n \to \mathcal{F} \leftarrow m$ *is a hypergraph with interfaces whose vertex labels are types in a signature* $\Sigma$*. The* input object *of the hypergraph is* $\widetilde{n} = \ell^*_{V,n}(V_n)$*, and the* output object *is* $\widetilde{m} = \ell^*_{V,m}(V_m)$*. For each edge* $e \in E_\mathcal{F}$*, the edge's* source object *is* $\widetilde{s(e)} = \ell^*_{V,\mathcal{F}}(s(e))$ *and its* target object *is* $\widetilde{t(e)} = \ell^*_{V,\mathcal{F}}(t(e))$*.*

We say these are source and target *objects* since these lists of types are objects in the free SMC on $\Sigma$. As examples, the input object of the hypergraph of Figure 5 is $A \otimes B \otimes A \otimes A$, the output object is $C \otimes A$, the source object of the edge labelled $f$ is $A \otimes B$, and the target object of the edge labelled $f$ is $C$.

▶ **Definition 7.** *Suppose* $\Sigma = (\Sigma_O, \Sigma_M, t)$ *is a signature and let* $(\Sigma_A, \Gamma)$ *be a set of variables with a context. An* mda-hypergraph (in $\Sigma$) *is a hypergraph with ordered interfaces* $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ *with labels from* $(\Sigma_O, \Sigma_M + \Sigma_A)$ *satisfying*

1. *directed acyclicity,*
2. *the in-degree and out-degree of every vertex of* $\mathcal{F}$ *is at most 1,*
3. *vertices of* $\mathcal{F}$ *with in-degree 0 are precisely the image of* $f$*,*
4. *vertices of* $\mathcal{F}$ *with out-degree 0 are precisely the image of* $g$*,*
5. *for all* $e \in E$ *with* $\ell_{E,\mathcal{F}}(e) \in \Sigma_M$*, we have* $\ell_{E,\mathcal{F}}(e) :_t \widetilde{s(e)} \to \widetilde{t(e)}$*, and*
6. *for all* $e \in E$ *with* $\ell_{E,\mathcal{F}}(e) \in \Sigma_A$*, we have* $f :_\Gamma: \widetilde{s(e)} \to \widetilde{t(e)}$*.*

*In this case, we say* $\mathcal{F}$ *is an mda-hypergraph* from the object $\widetilde{n}$ to the object $\widetilde{m}$*, and we say* $\widetilde{n}$ *and* $\widetilde{m}$ *are the* source and target objects *of* $\mathcal{F}$*, respectively. Two mda-hypergraphs with the same source and target objects are* parallel.

Cospans satisfying (2)–(4) are called *monogamous* [5, Definition 9]; "mda-hypergraph" is an abbreviation for "monogamous directed acyclic hypergraph". Conditions (5) and (6) are well-typing conditions. Condition (5) says if an edge is labelled by a constant, the input and output objects of that edge match the typing required by the signature. Condition (6) enforces the context's typing for the inputs and outputs of variable-labelled edges.

Two mda-hypergraphs in $\Sigma$ may have different variables or different contexts for those variables. We say that two mda-hypergraphs are *compatible* if their contexts give the same type to the variables they have in common.

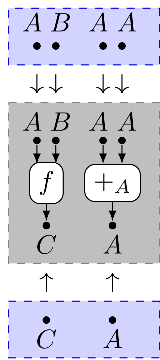



**Figure 5** Example mda-hypergraph.

The hypergraph of Figure 4a fails to be an mda-hypergraph on at least three counts. It does not have interfaces, the vertex labelled $A \otimes C$ has in-degree 2, and there is a directed cycle. The hypergraphs with interfaces of Figure 4b also are not mda-hypergraphs. The left fails a monogamy condition: there is a vertex with out-degree 0 that is not in the image of the output interface. The right fails a well-typedness condition since the morphism $\sigma_{A,B}$ has codomain $B \otimes A$ but the target vertex labels of the edge labelled $\sigma_{A,B}$ is $A \otimes B$.

The hypergraph depicted in Figure 5 is an mda-hypergraph. Further positive examples of mda-hypergraphs can be found in Figure 6.

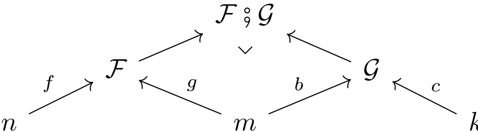
Since we intend to use mda-hypergraphs to represent morphisms, it is not surprising that these hypergraphs can be composed in various ways. Parallel composition is easiest.

▶ **Definition 8.** *If $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ and $h \xrightarrow{b} \mathcal{G} \xleftarrow{c} k$ are compatible mda-hypergraphs in $\Sigma$, their* parallel composition *$\mathcal{F} \boxtimes \mathcal{G}$ is the mda-hypergraph $n+h \xrightarrow{f+b} \mathcal{F}+\mathcal{G} \xleftarrow{g+c} m+k$. The ordering on $n+h$ has all the elements of $n$ before all the elements of $h$ and the given orderings within $n$ and $h$. Similarly for $m+k$.*

▶ **Lemma 9.** *Suppose $\mathcal{F}$ and $\mathcal{G}$ are mda-hypergraphs in $\Sigma$. If the source and target objects of $\mathcal{F}$ are $X$ and $Y$ and the source and targets of $\mathcal{G}$ are $Z$ and $W$, then the source and target of $\mathcal{F} \boxtimes \mathcal{G}$ are $X \otimes Z$ and $Y \otimes W$.*

Mda-hypergraphs can also be composed in sequence using pushouts.

▶ **Definition 10.** *If $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ and $m \xrightarrow{b} \mathcal{G} \xleftarrow{c} k$ are mda-hypergraphs in $\mathcal{C}$, their* sequential composition *$\mathcal{F} \,;\, \mathcal{G}$ is an mda-hypergraph formed by the pushout*



In more detail, this pushout is the quotient of disjoint union of the hypergraphs $\mathcal{F}$ and $\mathcal{G}$ where for each $v \in V_m$, we identify $g(v)$ in the copy of $\mathcal{F}$ with $b(v)$ in the copy of $\mathcal{G}$. The monogamy conditions ensure that after the output vertices of $\mathcal{F}$ are identified with the input vertices of $\mathcal{G}$, the resulting hypergraph again has the monogamy property.

▶ **Lemma 11.** *Suppose $\mathcal{F}$ and $\mathcal{G}$ are mda-hypergraphs in a signature $\Sigma$. If the source and target objects of $\mathcal{F}$ are $X$ and $Y$ and the source and targets of $\mathcal{G}$ are $Y$ and $Z$, then the source and target of $\mathcal{F} \,;\, \mathcal{G}$ are $X$ and $Z$.*

When considering non-freely generated monoidal categories, the labels of hypergraphs may involve tensor products of objects. In such cases, hypergraphs with a common object may not have composable interfaces. For this, an adapter hypergraph can be used.

▶ **Definition 12.** *Suppose $n$ and $m$ are two discrete ordered hypergraphs with the property that $\widetilde{n} = \widetilde{m}$. The $n, m$-*adapter *is the mda-hypergraph with a single edge $e$ labelled $1_A$ and vertices $n + m$ with $s(e) = n$, $t(e) = m$ with vertex labels matching the corresponding vertices in $n$ and $m$.*

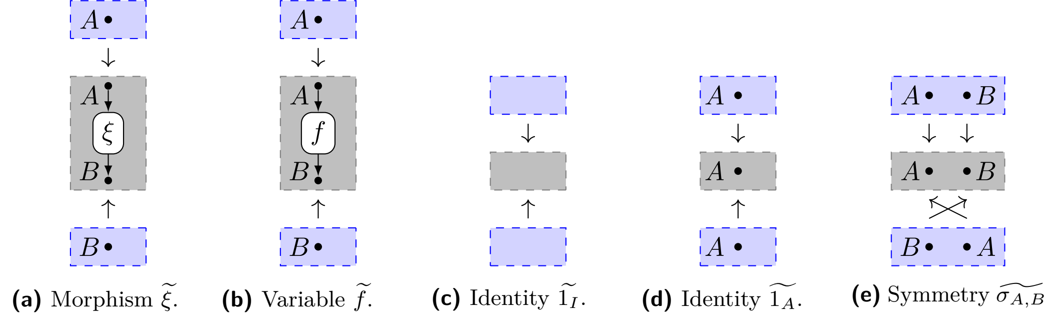
This solves the wire-matching problem in non-freely generated monoidal categories, but we continue to assume our categories are generated from a signature for clarity.

## 3.3 Mda-hypergraphs of morphism terms

Now we can define an mda-hypergraph interpretation for every morphism term.

▶ **Definition 13.** *Suppose $\tau$ is a morphism term in the signature $(\Sigma_O, \Sigma_M)$ with context $(\Sigma_A, \Gamma)$. The* mda-hypergraph interpretation of *$\tau$ is denoted $\widetilde{\tau}$ and is defined by induction.*
- *$\widetilde{\xi}$ where $\xi :_t A \to B$ is a constant is as in Figure 6a,*
- *$\widetilde{f}$ where $f :_\Gamma A \to B$ is a variable is as in Figure 6b,*
- *$\widetilde{1_I}$, $\widetilde{1_A}$, and $\widetilde{\sigma_{A,B}}$ are as in Figure 6c-6e,*
- *$\widetilde{\tau_1 \otimes \tau_2} = \widetilde{\tau_1} \boxtimes \widetilde{\tau_2}$, and $\widetilde{\tau_1 ; \tau_2} = \widetilde{\tau_1} \fatsemi \widetilde{\tau_2}$.*

*If $\mathcal{F}$ is an mda-hypergraph of a ground term (i.e. without variables), then we call it a* ground mda-hypergraph*.*




(a) Morphism $\widetilde{\xi}$. (b) Variable $\widetilde{f}$. (c) Identity $\widetilde{1_I}$. (d) Identity $\widetilde{1_A}$. (e) Symmetry $\widetilde{\sigma_{A,B}}$.

**Figure 6** Interpretation of morphism terms as mda-hypergraphs.

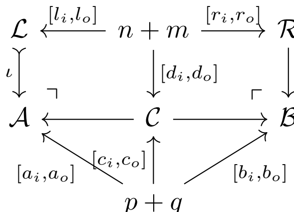
▶ **Lemma 14.** *If $\alpha =_{SMC} \beta$, then $\widetilde{\alpha} = \widetilde{\beta}$. Consequently, there is a strict symmetric monoidal functor $[\![-]\!]$ from $S_\Sigma$ to $\Sigma$-labelled mda-hypergraphs.*

## 3.4 Double pushout rewriting for mda-hypergraphs

String diagrams and morphism terms both support rewriting systems to facilitate equational reasoning. Hypergraphs have a similar rewriting system based on *double-pushout rewriting*. We recall this for the case of mda-hypergraphs next.

▶ **Definition 15.** *An* mda rewrite rule *is a parallel pair of mda-hypergraphs $\mathcal{L}$ and $\mathcal{R}$ with interfaces $n$ and $m$. An* mda rewrite system *is a set of rewrite rules. We say that the mda-hypergraph $\mathcal{A}$ with interfaces $p$ and $q$* rewrites directly *to the parallel hypergraph $\mathcal{B}$ if*

- *there is a mono $\iota : \mathcal{L} \to \mathcal{A}$ whose image is a convex subgraph of $\mathcal{A}$,*
- *there is an mda-hypergraph $p + m \xrightarrow{[c_i,d_o]} \mathcal{C} \xleftarrow{[c_o,d_i]} q + n$, and*
- *the diagram below commutes and the two marked squares are pushouts.*



As an example, consider an (instance of an) equation for a final map: $f;\omega_B = \omega_A$. An mda-rewrite turning $\widetilde{f;\omega_B}$ into $\widetilde{\omega_A}$ in $\delta_A; g \otimes \widetilde{(f;\omega_B)}$ is shown in Figure 7.

The mda-hypergraph $\mathcal{C}$ is $\mathcal{A}$ with the image of $\mathcal{L}$ deleted (except $\mathcal{L}$'s interface vertices). Inputs to $\mathcal{L}$ are then outputs of $\mathcal{C}$ and similarly outputs of $\mathcal{L}$ are inputs to $\mathcal{C}$. The explains the peculiar mixing of inputs and outputs in the cospan $p + m \xrightarrow{[c_i,d_o]} \mathcal{C} \xleftarrow{[c_o,d_i]} q + n$. In [5, Definition 23], $\mathcal{C}$ is called a *boundary complement*, a strengthening of the notion of *pushout complement*.

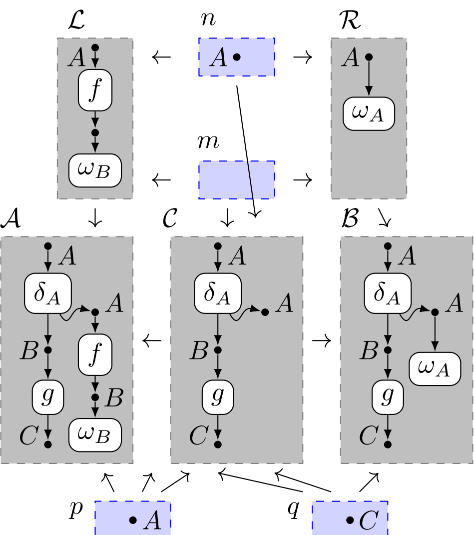


**Figure 7** Example rewrite step.

The connection between string diagram rewriting and (convex) double-pushout rewriting of ground mda-hypergraphs is thoroughly examined in [7, 5, 6]. As long as the convex embedding $\iota : \mathcal{L} \to \mathcal{A}$ exists, this DPO rewriting step can be completed uniquely (up to isomorphism).

Connecting term rewriting to double-pushout rewriting is similiarly possible, and requires only a treatment of substitution. If $f$ is a variable in the $\Sigma$-term $\alpha$, then $\widetilde{f}$ is an edge in $\widetilde{\alpha}$. Since this is a convex subgraph of $\widetilde{\alpha}$, we can use DPO rewriting to replace an edge labelled $f$ with a parallel hypergraph $\widetilde{\beta}$. The substitution $\widetilde{\alpha[\beta/f]}$ is then formed by subtituting all edges labelled $f$ in $\widetilde{\alpha}$ with $\widetilde{\beta}$ in this manner. This allows the substitutions required both to find substitution instances of rewrite rules and to create contexts in rewrite steps in DPO systems.

## 4 Hierarchical string diagrams

In this section, we embark on a similar project for hierarchical string diagrams for monoidal closed categories. We devise a suitable combinatorial structure, called *hypernets*, and show that two closed morphism terms are interpreted as the same hypernet whenever they are equivalent modulo the laws of symmetric monoidal categories (Proposition 24). This allows us to conclude that rewriting can be "implemented" as double-pushout rewriting of hypernets (Proposition 26).

Hierarchical hypergraphs have been used before, see e.g. [10, 15, 31]. Our approach is broadly similar, but with enough subtle differences that it is necessary to give our own definitions. For a more detailed comparison, see section 6.2.

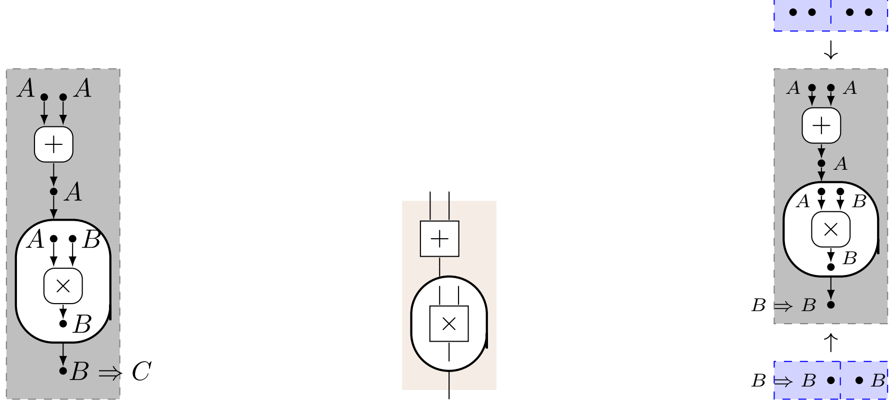
### 4.1 Hierarchical hypergraphs and hypernets

A hierarchical hypergraph is a hypergraph with an extra parent relationship determining the hierarchical structure.

▶ **Definition 16.** *A hierarchical hypergraph is a tuple $(V, E, s, t, \ell_V, \ell_E, p_V, p_E)$ where $V, E, s, t,$ and $\ell_V$ are as in a hypergraph, the edge labelling is modified to include an extra value $\ell_E : E \to \Sigma_E + 1$, and parent functions $p_V : V \to E + 1$ and $p_E : E \to E + 1$.*

*The parent functions satisfy some conditions. First, an edge and any of its source and target vertices must have the same parent: $p_V(v) = p_E(e) = p_V(v')$ for all $v \in s(e)$ and $v' \in t(e)$, respectively. Second, the parent relation must be acyclic. More precisely, we assume for all $e \in E$ there is some $k \geq 1$ such that $(p_{E,\perp})^k(e) = \perp$ where $\perp$ is the element of 1 and $p_{E,\perp} : E + 1 \to E + 1$ is the extension of $p_E$ adding $p_{E,\perp}(\perp) = \perp$.*

When the parent of a vertex or edge is the element $\perp$ from the right summand, we say it is an *outermost vertex* or *outermost edge*. If the label of an edge is $\perp$, we say (with some abuse) that it is *unlabelled*. When considering multiple hierarchical hypergraphs, we use subscripts to disambiguate these data.




**(a)** Hierarchical hypergraph. **(b)** Hierarchical string diagram. **(c)** HHG with interfaces.

**Figure 8** Example hierarchical structures.

In every hierarchical hypergraph $\mathcal{F}$, associated to every edge $\hat{e}$ is a subgraph, namely the subgraph of edges $e$ (and vertices $v$) satisfying $p^k_{E,\perp}(e) = \hat{e}$ (and $(p_{E,\perp})^j(p_V(v)) = \hat{e}$) for some $k \geq 1$ (and $j \geq 0$). We denote this subgraph $\mathcal{F}_{\hat{e}}$ and call it "the inner hypergraph of $\hat{e}$".

When depicting a hierarchical hypergraph, we indicate the inner hypergraph of an edge by nesting the inner subgraph within its edge, like abstraction in hierarchical string diagrams. If a subgraph $\mathcal{G}$ of a hierarchical hypergraph $\mathcal{F}$ has the property that $\mathcal{F}_{\hat{e}} \subseteq \mathcal{G}$ for all $\hat{e} \in E_{\mathcal{G}}$, we call $\mathcal{G}$ *down-closed*.

An example hierarchical hypergraph is shown in Figure 8a. This hierarchical hypergraph has 3 edges, labelled $+$, $\times$, and one unlabelled. The unlabelled edge is the parent of the edge labelled $\times$ (and its sources and targets), and so the $\times$ edge is depicted inside. This notation echoes the bubble operation in hierarchical string diagrams. The corresponding hierarchical string diagram for this hypergraph is depicted in Figure 8b.

▶ **Definition 17.** *A* morphism of hierarchical hypergraphs $\phi : \mathcal{F} \to \mathcal{G}$ *is a pair of functions* $\phi_V : V_{\mathcal{F}} \to V_{\mathcal{G}}$ *and* $\phi_E : E_{\mathcal{F}} \to E_{\mathcal{G}}$*, which is a morphism of the underlying hypergraphs and respects the hierarchial structure in the following sense:*

1. $(p_{V,\mathcal{F}}; \phi_E)(v) = (\phi_V; p_{V,\mathcal{G}})(v)$ *if* $p_{V,\mathcal{F}}(v) \in E_{\mathcal{F}}$
2. $(p_{E,\mathcal{F}}; \phi_E)(e) = (\phi_E; p_{E,\mathcal{G}})(e)$ *if* $p_{E,\mathcal{F}}(e) \in E_{\mathcal{F}}$

Note that we do not require that outermost vertices and edges are sent to outermost vertices and edges. If $\phi_V$ and $\phi_E$ additionally satisfy the property that $p_{V,\mathcal{G}}(\phi_V(v)) = \bot$ and $p_{E,\mathcal{G}}(\phi_E(e)) = \bot$ whenever $p_{V,\mathcal{F}}(v) = \bot$ and $p_{E,\mathcal{G}}(e) = \bot$, then the morphism is *strict*.

Hierarchical hypergraphs and the morphisms between them form a category. A hierarchical hypergraph *with interfaces* is a cospan in this category, $n \xrightarrow{f} \mathcal{H} \xleftarrow{g} m$ with $n$ and $m$ discrete and ordered. The *input interface of the edge* $e \in E_{\mathcal{H}}$ is the subgraph of vertices $v$ of $n$ such that $(f; p_{V,\mathcal{H}})(v) = e$. Similarly, the *output interface of the edge* $e$ is the subgraph of $m$ satsifying $(g; p_{V,\mathcal{H}})(v) = e$. The *outermost input and output interfaces* are the subgraphs of the interfaces whose image has parent $\bot$.

When labels from a hierarchical hypergraph are drawn from a signature, we define the input and output **objects** for each of these interfaces as in Definition 6: the list of labels of the vertices in the interface in the same order as the vertices.

An example hierarchical hypergraph with interfaces is shown in Figure 8c. Note that the input interface has the $B$-labelled source vertex of the $\times$ edge in its image, and the output interface similarly includes the $B$-labelled target vertex. These are *internal interfaces* for the hierarchical hypergraph, since they are not outermost vertices. We separate the outermost interface and internal interfaces in our graphical depiction with a vertical line.

▶ **Definition 18.** *Suppose* $n \xrightarrow{f} \mathcal{H} \xleftarrow{g} m$ *is a* $\Sigma$*-labelled hierarchical hypergraph with interfaces. Let* $e \in E_{\mathcal{H}}$ *be an edge with* $\ell_E(e) = \bot$*, let* $P$ *be the input object for the input interface of* $e$*, and let* $C$ *be the output object for the output interface of* $e$*. We say* $e$ *is a* well-typed abstraction *if there is an object* $B$ *such that* $\widetilde{s(e)} \otimes B = P$ *and* $\widetilde{t(e)} = B \Rightarrow C$*.*

Note also the difference between the "input/output object of an edge", which considers the edge's sources and targets, and the "input/output object for the interface of an edge", which considers the dangling vertices whose parent is that edge.

As examples, the unlabelled edge in Figure 8a is not a well-typed abstraction, assuming $A$, $B$ and $C$ are distinct generators. Its input, output, input interface, and output interface objects are $A$, $B \Rightarrow C$, $A \otimes B$, and $B$, respectively. There is no object $X$ such that $A \otimes X = A \otimes B$ and $B \Rightarrow C = X \Rightarrow B$, so it is not a well-typed abstraction. On the other hand, the unlabelled edge in Figure 8c is a well-typed abstraction: its output object has been changed to $B \Rightarrow B$, so clearly $X = B$ is an object satisfying the necessary conditions.

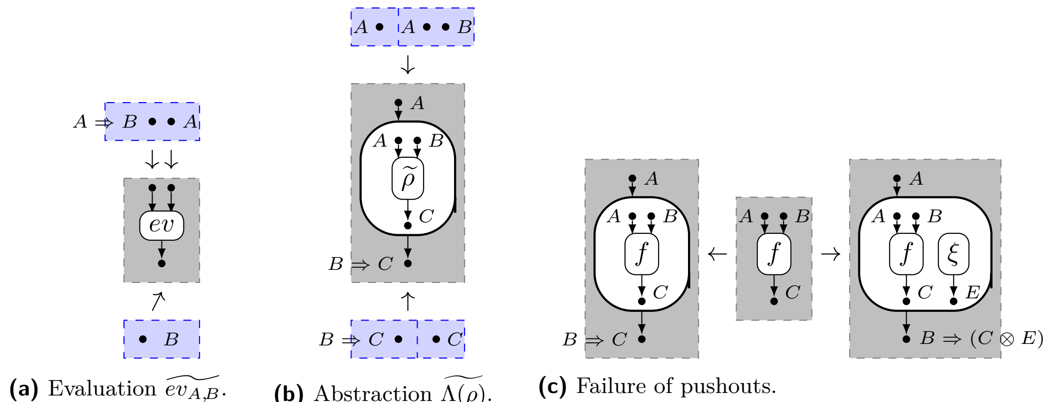
Hierarchical hypergraphs satisfying this well-typedness condition are our formal model for hierarchical string diagrams. We call members of this restricted class *hypernets*.

▶ **Definition 19.** *A* hypernet *is a hierarchical hypergraph $n \to \mathcal{H} \leftarrow m$ with interfaces such that (1) it is an mda-hypergraph when the hierarchical structure is forgotten, (2) if $\ell_E(e) \neq \bot$, then $\mathcal{H}_e$ is the empty hypergraph, and (3) if $\ell_E(e) = \bot$, then $e$ is a well-typed abstraction.*

*If a hierarchical hypergraph only satisfies properties (1) and (2) only, we call it a* weak hypernet*.*

Hypernets can be composed in parallel just like mda-hypergraphs with interfaces. They can be composed in sequence anytime their *outermost* interfaces match again using a pushout.

▶ **Definition 20.** *Suppose $\rho$ is a closed morphism term in a signature $\Sigma$. The* hypernet interpretation *of $\rho$ is denoted $\widetilde{\rho}$, and again defined by induction on $\rho$. The cases shared in common with morphism terms are as in Definition 13. The two new cases are ev and $\Lambda(\rho)$, which are defined as in Figure 9a and 9b.*



**(a)** Evaluation $\widetilde{ev_{A,B}}$. **(b)** Abstraction $\widetilde{\Lambda(\rho)}$. **(c)** Failure of pushouts.

**Figure 9** Hypernet interpretations and pushouts.

## 4.2 Pushout rewriting of hypernets

We next consider pushouts in order to support double pushout rewriting. When allowing all hierarchical hypergraph morphisms, the category of hierarchical hypergraphs does not have all pushouts or even pushouts along monos. This is due to ambiguities in the parents of outermost vertices and edges. Two non-strict morphisms can embed a graph into two unmergeable parts of different graphs. As an example, the cospan of Figure 9c does not have a pushout, even if we allow non-well-typed abstractions.

Consequently, much of the advanced categorical structure of hypergraphs which have historically proven useful, such as the fact they form an adhesive category, cannot be used when studying hierarchical hypergraphs or hypernets. However, the pushouts we need to exist still exist.

As a running example, we will consider a hypernet rewriting rule corresponding to the slide rule of Figure 3c. The hypernets corresponding to each of these terms are shown in Figure 10. The most obvious difference between this span and a rewriting span for mda-hypergraphs is that the interfaces for the left and right legs do not exactly match! In fact, we only need the outermost interfaces to match: the interior interfaces are only important for enforcing the well-typedness of abstractions.

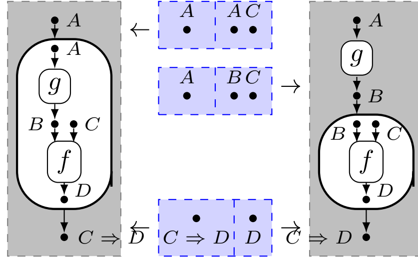
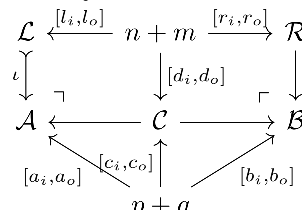



**Figure 10** A hypernet span for the slide rule.

We have generally been using terms for hypergraphs also for hierarchical hypergraphs. However, we must refine two important defintions for rewriting hypernets. We say that two hierarchical hypergraphs are *parallel* if they have the same *outermost* input and output interfaces. We say a subgraph is *convex* (in a larger hierarchical hypergraph) if it is convex (as a hypergraph) **and** down-closed.

▶ **Definition 21.** *A* hypernet rewrite rule *is a parallel pair of hypernets $\mathcal{L}$ and $\mathcal{R}$ with outermost interfaces $n$ and $m$. We say that the hypernet $\mathcal{A}$ with interfaces $\mathbf{p}$ and $\mathbf{q}$ and outermost interfaces $p$ and $q$* rewrites directly *to the parallel hypernet $\mathcal{B}$ if*

- *there is a mono $\iota : \mathcal{L} \to \mathcal{A}$ whose image is a convex subgraph of $\mathcal{A}$,*
- *there is a weak hypernet $\mathbf{p} + m \xrightarrow{[\mathbf{c_i}, d_o]} \mathcal{C} \xleftarrow{[\mathbf{c_o}, d_i]} \mathbf{q} + n$, and*
- *the diagram below commutes and the two marked squares are pushouts.*



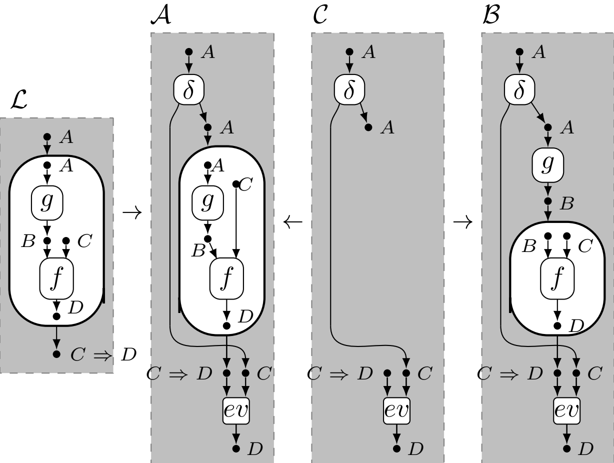




**Figure 11** A rewriting span.

The primary difference between hypernet rewriting and mda-hypergraph rewriting (Definition 15) is that the diagram for hypergraph rewriting only places conditions on the outermost interfaces, rather than the entire interface.

Next we give an example application of the hypernet rewrite rule of Figure 10. The leftmost morphism in Figure 11 is a convex embedding of the leftmost hypernet in the rewrite rule, $\mathcal{L}$, into a larger hypernet, $\mathcal{A}$. The next hypernet, $\mathcal{C}$, is the pushout complement of $\mathcal{L}$ in $\mathcal{A}$. Note that the outermost interface of $\mathcal{C}$ consists of the outermost interface of $\mathcal{A}$ together with the outermost interface of $\mathcal{L}$. This occurs exactly when the embedding morphism of the left side of the rule is strict (sends outermost to outermost). If this matching were not strict (sending $\mathcal{L}$ to an interior hypergraph), the outermost interface of $\mathcal{L}$ would still be part of the interface of $\mathcal{C}$, but would be an internal interface. Note also that internal interfaces in $\mathcal{L}$ are not part of the interface of $\mathcal{C}$, since all of $\mathcal{L}$ (except its interface vertices) are deleted in the pushout complement. Finally, $\mathcal{B}$ is the pushout of $\mathcal{C}$ with $\mathcal{R}$ along their common interfaces.

▶ **Proposition 22.** *Suppose $\langle \mathcal{L}, \mathcal{R} \rangle$ is a hypernet rewriting rule, $\mathcal{A}$ is a hypernet, and $\iota : \mathcal{L} \to \mathcal{A}$ is a mono with a convex image in $\mathcal{A}$. Then the boundary complement $\mathcal{C}$ of Definition 21 exists, and further the pushout $\mathcal{B}$ is a hypernet.*

**Proof (Idea).** The boundary complement is constructed as in hypergraphs. The unusual part is that the boundary complement can fail to be a hypernet. This happens precisely when the embedding $\iota$ is not strict. In this case, new interfaces are created inside an edge; that edge fails to be a well-typed abstraction so $\mathcal{C}$ will only be a weak hypernet. Fortunately, takng the pushout with a hypernet $\mathcal{R}$ with the same outermost interface as $\mathcal{L}$ restores the well-typedness of the abstraction. ◀

## 5 Soundness and completeness

To describe the connection between term rewriting and hypernet rewriting, we first note that the hypernets corresponding to the subterms of a term are always convex subgraphs of the hypernet for the full term. The converse is not generally true: $\widetilde{f \otimes h}$ is a convex subgraph of $\widetilde{f \otimes g \otimes h}$, but $f \otimes h$ is not a subterm of $f \otimes g \otimes h$, regardless of how the latter is constructed. Note, however, that there is a term equivalent to $f \otimes g \otimes h$ under SMC equations, namely $1 \otimes \sigma; f \otimes h \otimes g; 1 \otimes \sigma$, for which $f \otimes h$ does appear. This motivates our next definition and result.

▶ **Definition 23.** *A closed morphism term $\rho$ is a* possible subterm *of another closed morphism term $\alpha$ if there is a closed morphism term $\beta =_{SMC} \alpha$ such that $\rho$ is a subterm of $\beta$.*

▶ **Proposition 24.** *For every closed morphism term $\rho$, possible subterms of $\rho$ and (isomorphism classes of) convex sub-hypernets of $\widetilde{\rho}$ are in bijective correspondence.*

In particular, this implies that every hypernet has exactly one SMC equivalence class of terms representing it.

The next critical notion is that of *substitution*. In terms, substitution replaces all instances of a variable in a term with another term. Hypernet rewriting, however, may not be able to mimic term substitution as a *single* rewrite if the subgraph of edges labelled with this variable do not form a convex subgraph. However, it can always be accomplished in several steps.

▶ **Lemma 25.** *Suppose $\alpha$ and $\beta$ are closed morphism terms and $f$ is a variable. There is a finite sequence of hypergraphs $\mathcal{H}_0, \ldots, \mathcal{H}_n$ such that $\widetilde{\alpha} = \mathcal{H}_0$, $\widetilde{\alpha[\beta/f]} = \mathcal{H}_n$ and $\mathcal{H}_i$ directly rewrites to $\mathcal{H}_{i+1}$ under the rule $\langle \widetilde{f}, \widetilde{\beta} \rangle$.*

In such a case, we say $\mathcal{H}_n$ is a *substitution instance* of $\mathcal{H}_0$. Finally, we can formalize the correspondence between term rewriting and hypernet rewriting.

▶ **Proposition 26.** *Suppose $\alpha, \beta, \lambda$, and $\rho$ are closed morphism terms such that $\alpha \to_{\langle\lambda,\rho\rangle} \beta$. There are hypernets $\mathcal{L}$ and $\mathcal{R}$, substitution instances of $\widetilde{\lambda}$ and $\widetilde{\rho}$ respectively, such that $\widetilde{\alpha} \Rightarrow_{\langle\mathcal{L},\mathcal{R}\rangle} \widetilde{\beta}$.*

▶ **Proposition 27.** *Suppose $\alpha, \beta, \lambda$, and $\rho$ are closed morphism terms such that $\widetilde{\alpha} \Rightarrow_{\langle\widetilde{\lambda},\widetilde{\rho}\rangle} \widetilde{\beta}$. There are closed morphism terms $\gamma, \delta$ such that $\alpha =_{SMC} \gamma$ and $\beta =_{SMC} \delta$ and $\gamma \to_{\langle\lambda,\rho\rangle} \delta$.*

These propositions, taken together, show an equivalence of expressiveness: every step in one rewriting system can be accomplished in the other (though potentially in many steps). Term rewriting has the advantage that substitution instances can be formed in a single operation, where imitating this in hypernets requires each instance of the variable be replaced individually. However, hypernet rewriting has the advantage that it is able to replace *possible subterms* from a context, where term rewriting may need to do SMC rewriting steps first to realize the possible subterm as an actual subterm before replacing.

# 6 Related work

## 6.1 String diagrams

Monoidal closed categories have been thoroughly studied in the context of logic and type theory, because of the well-known correspondence of their internal language with (linear) simply typed $\lambda$-calculus and linear logic [35, 19].

To the best of our knowledge, we provide the first fully specified string diagrammatic language for closed categories. Our approach shares similarities with the formalisms of sharing graphs for describing $\lambda$-calculus computations [25]. The main difference is that string diagrams, albeit graphical in appearance, can be manipulated as a syntax, whereas sharing graphs are usually studied as combinatorial objects. Unlike syntax, reasoning about graphs algebraically requires a higher degree of technical sophistication [20]. Finally, sharing graphs are typically used to study low-level computational models for functional languages, in particular quantitative models [29], whereas our approach is more focussed on equational reasoning and rewriting, and does not have the ambition of investigating the resources employed during computation.

Monoidal closed categories also extend to $\star$-autonomous categories. These are relevant to the study of multiplicative linear logic and have been extensively studied in terms of proof nets. Our graphical calculus is essentially different from proof nets. The grammar of morphisms does not stem from a sequent calculus, and we capture the intended semantics via equations rather than a correctness criterion. But the connection might be made precise relying on the existing translations between proof nets and string diagrams [22, 34]. Finally, a different style of hierarchical string diagrams appear in the literature to represent universal properties graphically such as Kan extensions [21] and free monads [32].

The only other proposal for a string-diagram language for monoidal closed categories which we are aware of is that of [3]. To keep the language of types as simple as possible and as *strict* as possible they propose an intriguing graphical innovation, a so-called *clasp* operator on stems. The exponential type is represented using the clasp, and much like in our own language, a *bubble* is used to represent currying.

Although not presented explicitly as a string diagram language, the treatment of closures in [33] is related in methodology to our work, although the setting of *partially-traced partially-closed premonoidal categories* is significantly different to ours.

## 6.2 Hierarchical hypergraphs and rewriting

The notion of hierarchical hypergraph used in this paper is inspired by and a formalisation of the graphs used in [18].

Although there is no consensus on a standard definition of hierarchical graphs, the various approaches to these structures [10, 15, 31] give slight variations on the idea of graphs containing other graphs and notions of morphisms between them. Some of the differences are minor – ours are directed, others [15] are not – but other differences are fundamental. Sometimes edges are permitted to connect vertices with different parents, as in [31, 10], sometimes this is prohibited (as it is here). Some approaches consider only strict morphisms, but others relax the notion of morphism. Due to the subtle but technically significant differences between our requirements and the properties of previous works, it was not possible to reuse previous work wholesale, and we found it necessary to introduce our own variation.

The formal correspondence between monoidal closed categories and hierarchical hypergraphs lies in a tradition of analogous results relating string diagram rewriting and double-pushout hypergraph rewriting, see [7, 5, 6]. To the best of our knowledge, such correspondence has not been spelled out in the way presented in our work, although the idea of linking the exponential structure of closed categories with the hierarchy structure of hierarchical hypergraphs may be found in [13]. Although it does not uses string diagrams or other categorical tools, the algebraic specification language for hierarchical graphs studied in [9] is aiming towards similar goals. Representing intermediate stages of the compiler as graphs is a long-established practice in compiler design and engineering. Graphs are an *efficient* syntactic representation which are recognised as a better target for optimisation and analysis than raw text. In its simplest incarnation the graph representation of terms is just an *abstract syntax tree*, but more sophisticated representations were increasingly used [12], sometimes leading to specific and novel optimisation techniques [30].

The use of graph-like representation outside of compiler engineering has a lot of untapped potential, as advocated by some [16]. This is not entirely new, for example interaction nets are a graph-like semantics of higher-order computation [24], albeit highly informal.

We would also like to point out recent work on $\mathcal{M},\mathcal{N}$-adhesivity that can be used with hierarchical graphs [11]. Though our category of hierarchical hypergraphs is not adhesive, it is possible that choosing a different family of morphisms could make it $\mathcal{M},\mathcal{N}$-adhesive. For example, we believe this category with strict morphisms only is adhesive.

Finally, another related line of work which we found inspirational is the use of graph-like languages inspired by proof nets to bridge the gap between syntax and abstract machines, in order to provide a quantitative analysis of reduction strategies for the lambda calculus [1].

# 7 Conclusion and further work

In this paper, we have presented a hypergraph-based formalism for representing string diagrams of monoidal closed categories. Our approach is based on interpreting morphism terms as hypernets and comparing their rewriting systems. This makes it easy to express universally quantified equations while retaining a simple redex search. An alternative approach, closer in spirit to [7, 5, 6], would be to interpret the morphisms of the (closed) monoidal category instead. This would make the interpretation functorial and has further computational complexity benefits; we plan to develop this connection between hypernet and hierarchical string diagram rewriting in further work.

## References


1. Beniamino Accattoli. Proof nets and the call-by-value $\lambda$-calculus. *Theor. Comput. Sci.*, 606:2–24, 2015. doi:10.1016/j.tcs.2015.08.006.
2. Mario Alvarez-Picallo, Dan R Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *arXiv preprint arXiv:2107.13433*, 2021.
3. John Baez and Mike Stay. Physics, topology, logic and computation: a rosetta stone. In *New structures for physics*, pages 95–172. Springer, 2010. doi:10.1007/978-3-642-12821-9_2.
4. Guillaume Boisseau and Paweł Sobociński. String diagrammatic electrical circuit theory. *arXiv preprint arXiv:2106.07763*, 2021.
5. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory II: rewriting with symmetric monoidal structure. *CoRR*, abs/2104.14686, 2021.
6. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory III: confluence with and without Frobenius. *Mathematical Structures in Computer Science*, to appear, 2021.
7. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory I: rewriting with frobenius structure. *J. ACM*, 69(2):14:1–14:58, 2022.
8. Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. *ACM SIGPLAN Notices*, 50(1):515–526, 2015.
9. Roberto Bruni, Fabio Gadducci, and Alberto Lluch-Lafuente. An algebra of hierarchical graphs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2010. doi:10.1007/978-3-642-15640-3_14.
10. Roberto Bruni, Fabio Gadducci, and Alberto Lluch-Lafuente. An algebra of hierarchical graphs and its application to structural encoding. *Sci. Ann. Comput. Sci.*, 20:53–96, 2010. URL: http://www.info.uaic.ro/bin/Annals/Article?v=XX&a=2.
11. Davide Castelnovo, Fabio Gadducci, and Marino Miculan. A new criterion for $\mathcal{M},\mathcal{N}$-adhesivity, with an application to hierarchical graphs, 2022. doi:10.48550/ARXIV.2201.00233.
12. Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In Michael D. Ernst, editor, *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, pages 35–49. ACM, 1995. doi:10.1145/202529.202534.
13. Matteo Coccia, Fabio Gadducci, and Ugo Montanari. Gs.lambda theories: A syntax for higher-order graphs. In Richard Blute and Peter Selinger, editors, *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002*, volume 69 of *Electronic Notes in Theoretical Computer Science*, pages 83–100. Elsevier, 2002. doi:10.1016/S1571-0661(04)80560-7.
14. Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
15. Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002. doi:10.1006/jcss.2001.1790.
16. Dan R. Ghica. Operational semantics with hierarchical abstract syntax graphs. In Patrick Bahr, editor, *Proceedings 11th International Workshop on Computing with Terms and Graphs, TERMGRAPH@FSCD 2020, Online, 5th July 2020*, volume 334 of *EPTCS*, pages 1–10, 2020. doi:10.4204/EPTCS.334.1.
17. Dan R. Ghica, Achim Jung, and Aliaume Lopez. Diagrammatic semantics for digital circuits. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages

24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CSL.2017.24.

18 Dan R. Ghica, Koko Muroya, and Todd Waugh Ambridge. Local reasoning for robust observational equivalence. *CoRR*, abs/1907.01257, 2019. arXiv:1907.01257.

19 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

20 Stefano Guerrini. A general theory of sharing graphs. *Theor. Comput. Sci.*, 227(1-2):99–151, 1999. doi:10.1016/S0304-3975(99)00050-X.

21 Ralf Hinze. Kan extensions for program optimisation or: Art and dan explain an old trick. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, volume 7342 of *Lecture Notes in Computer Science*, pages 324–362. Springer, 2012. doi:10.1007/978-3-642-31113-0_16.

22 Dominic Hughes. Simple free star-autonomous categories and full coherence. *CoRR*, 2005. URL: https://arxiv.org/abs/math/0506521.

23 André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, July 1991. doi:10.1016/0001-8708(91)90003-P.

24 Yves Lafont. Interaction nets. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 95–108. ACM Press, 1990. doi:10.1145/96709.96718.

25 John Lamping. An algorithm for optimal lambda calculus reduction. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 16–30. ACM Press, 1990. doi:10.1145/96709.96711.

26 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

27 Paul-André Melliès. Functorial boxes in string diagrams. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2006. doi:10.1007/11874683_1.

28 José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990. doi:10.1016/0890-5401(90)90013-8.

29 Koko Muroya and Dan R. Ghica. The dynamic geometry of interaction machine: A token-guided graph rewriter. *Log. Methods Comput. Sci.*, 15(4), 2019. doi:10.23638/LMCS-15(4:7)2019.

30 Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. 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 31–44. ACM, 2020. doi:10.1145/3385412.3386012.

31 Wojciech Palacz. Algebraic hierarchical graph transformation. *J. Comput. Syst. Sci.*, 68(3):497–520, 2004. doi:10.1016/S0022-0000(03)00064-3.

32 Maciej Piróg and Nicolas Wu. String diagrams for free monads (functional pearl). In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 490–501. ACM, 2016. doi:10.1145/2951913.2951947.

33 Ralf Schweimeier and Alan Jeffrey. A categorical and graphical treatment of closure conversion. In Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov, editors, *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 481–511. Elsevier, 1999. doi:10.1016/S1571-0661(04)80090-2.

34 Michael Shulman. $*$-autonomous envelopes and 2-conservativity of duals. *CoRR*, 2020. arXiv:2004.08487.

35 Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism.* Elsevier, 2006.

36 David Sprunger and Shin-ya Katsumata. Differentiable causal computations via delayed trace. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2019.

# Stateful Structural Operational Semantics

**Sergey Goncharov**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

**Stefan Milius**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

**Lutz Schröder**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

**Stelios Tsampas**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

**Henning Urbat**
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

**Abstract**

Compositionality of denotational semantics is an important concern in programming semantics. Mathematical operational semantics in the sense of Turi and Plotkin guarantees compositionality, but seen from the point of view of stateful computation it applies only to very fine-grained equivalences that essentially assume unrestricted interference by the environment between any two statements. We introduce the more restrictive *stateful SOS* rule format for stateful languages. We show that compositionality of two more coarse-grained semantics, respectively given by assuming read-only interference or no interference between steps, remains an undecidable property even for stateful SOS. However, further restricting the rule format in a manner inspired by the *cool* GSOS formats of Bloom and van Glabbeek, we obtain the *streamlined* and *cool* stateful SOS formats, which respectively guarantee compositionality of the two more abstract equivalences.

**2012 ACM Subject Classification** Theory of computation → Categorical semantics

**Keywords and phrases** Structural Operational Semantics, Rule Formats, Distributive Laws

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.30

**Related Version** *Extended paper with full proofs*: https://arxiv.org/abs/2202.10866

**Funding** *Sergey Goncharov*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 215418801.
*Stefan Milius*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 259234802.
*Lutz Schröder*: Work supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 (grant number 393541319/GRK2475/1-2019).
*Stelios Tsampas*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 419850228.
*Henning Urbat*: Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 419850228.

## 1 Introduction

A key prerequisite for modular reasoning about process calculi and programming languages is *compositionality*: A denotational semantics is compositional if the associated semantic equivalence forms a congruence, that is, subterms of a given process or program term may be replaced with equivalent subterms without affecting the overall denotational meaning of the term. For instance, the classical GSOS format of Bloom et al. [8] provides a unified formal

© Sergey Goncharov, Stefan Milius, Lutz Schröder, Stelios Tsampas, and Henning Urbat;
licensed under Creative Commons License CC-BY 4.0

representation of process languages interpreted over non-deterministic labelled transition systems, and guarantees that bisimilarity is compositional. Similarly, syntactic restrictions of the GSOS format due to Bloom [7] and van Glabbeek [39] guarantee compositionality for coarser equivalences.

More abstractly, GSOS is captured in Turi and Plotkin's bialgebraic framework of *mathematical operational semantics* [38], in which sets of operational semantic rules are represented as distributive laws of a monad over a comonad, a principle that has come to be used in widely varying semantic settings [5, 21, 12, 23]. In particular, Turi and Plotkin demonstrated that GSOS rules correspond precisely to natural transformations of type

$$\varrho_X \colon \Sigma(X \times (\mathscr{P}_\omega X)^L) \to (\mathscr{P}_\omega \Sigma^\star X)^L,$$

where $\Sigma$ is a polynomial functor on the category of sets (representing the signature of the process language at hand), $L$ is a set of (transition) labels, $\mathscr{P}_\omega$ is the finite power set functor, corresponding to finitary non-determinism, and $\Sigma^\star$ denotes the free (term) monad on $\Sigma$. This is an instance of an *abstract GSOS law*, a natural transformation of type $\Sigma(\mathrm{Id} \times T) \Longrightarrow T\Sigma^\star$, with $T$, the *behaviour functor*, instantiated to the functor $\mathscr{P}_\omega^L$, which is associated with image-finite $L$-labelled transition systems.

There is long-standing interest in SOS style specifications of stateful programming languages [32]. The natural instantiation of mathematical operational semantics to this setting would use $TX = (S \times (X+1))^S$ as the behaviour functor (for a given set $S$ of *states*). This gives rise to an extremely expressive rule format: In abstract GSOS laws of type $\Sigma(\mathrm{Id} \times T) \Longrightarrow T\Sigma^\star$, program constructs receive their arguments as full-blown state transformers, which in particular they can execute or probe on any number of input states. The semantic domain provided by mathematical operational semantics in this case is the final coalgebra for $T$, which consists of possibly infinite $S$-branching, $S$-labelled trees, and thus is an instance of *(coalgebraic) resumption semantics* [29], originally developed for concurrent settings [13, 10]. The induced notion of semantic equivalence, for which the format guarantees compositionality, is very fine-grained: Being a resumption semantics, it assumes that programs cede complete control to the environment between any two consecutive steps, and thus makes rather few programs equivalent. Capturing less sceptical semantics, such as standard sequential end-to-end net execution, in a compositional manner has proved rather more challenging; generally speaking, compositionality is harder for coarser equivalences because less information is available about the behaviour of subterms [39].

In the present work, we approach this problem by restricting the rule format to various degrees. We first note that the operational rules typically associated to imperative languages resemble GSOS rules with an additional input parameter, the present state. We correspondingly introduce the *stateful SOS* format for the specification of stateful languages, and show that stateful SOS specifications are in an one-to-one correspondence with natural transformations of type

$$\delta_X \colon S \times \Sigma(X \times S \times (X+1)) \to S \times (\Sigma^\star X + 1).$$

In a small-step operational semantics given in terms of transitions on pairs consisting of states in $S$ and program terms (or a termination marker $\checkmark \in 1$), $\delta_X$ assigns to a given state (in $S$) and a program construct applied to argument variables with given next-step operational behaviour (i.e. an element of $\Sigma(X \times S \times (X+1))$) its small-step operational behaviour. Effectively, this means that, in small-step operational semantics, program constructs can execute and probe their arguments only on the current state. We give a resumption semantics (over the final coalgebra for $T$ as above) for stateful SOS, and show that this semantics agrees with the one obtained by converting $\delta$ into a GSOS law, in particular is compositional.

**Table 1** Separating denotational domains by program equivalences.

| | $(\mathtt{x} \coloneqq 1;\ \mathtt{y} \coloneqq \mathtt{x}) = (\mathtt{x} \coloneqq 1;\ \mathtt{y} \coloneqq 1)$ | $(\mathtt{x} \coloneqq 1;\ \mathtt{x} \coloneqq 2) = (\mathtt{x} \coloneqq 2)$ |
|---|---|---|
| $\nu\gamma.\,(S \times (\gamma + 1))^S$ | ✘ | ✘ |
| $(S^+ + S^\omega)^S$ | ✔ | ✘ |
| $(S + 1)^S$ | ✔ | ✔ |

We go on to define two successive coarsenings of resumption semantics: *Trace semantics* assumes that the environment can observe but not manipulate states reached in between successive computation steps, and correspondingly uses the semantic domain $(S^+ + S^\omega)^S$, the set of functions expecting an initial state and returning a possibly terminating $S$-stream. The, yet coarser, *termination semantics* additionally abstracts from the intermediate states of a computation, and thus is defined over the semantic domain $(S + 1)^S$, the set of functions expecting an initial state and returning either a final state or divergence. Trace semantics has been used, e.g., in the type-theoretic semantics of program logics [25] and in formalizing concurrent systems that feature memory isolation mechanisms [27, 28]. Termination semantics is the semantic domain typically associated with *big-step* [22, 26] or *natural* semantics [18], and is a popular choice in settings where fine architectural details are less relevant [33, 31, 30]. Table 1 presents the three domains in decreasing order of granularity and illustrates their differences in terms of the programs they distinguish. Here, $S$ is the set of variable stores assigning to every program variable its current value. First, consider the programs $\mathtt{x}{\coloneqq}1;\ \mathtt{x}{\coloneqq}2$ and $\mathtt{x} \coloneqq 2$. These are clearly equivalent in termination semantics but not in trace semantics, as the additional initial step of the first program is visible in trace semantics. Similarly, the programs $\mathtt{x} \coloneqq 1;\ \mathtt{y} \coloneqq \mathtt{x}$ and $\mathtt{x} \coloneqq 1;\ \mathtt{y} \coloneqq 1$ are clearly equivalent under trace semantics but not under resumption semantics, as the latter assumes that the value of x may be changed by the environment between the two steps. In fact, we show as our first main result that despite the restricted expressiveness, it is undecidable whether the coarser program equivalences are compositional for a given stateful SOS specification. In a subsequent step, we thus introduce two sets of syntactic restrictions in the spirit of Bloom [7] and van Glabbeek [39], and show that these guarantee that stateful SOS specifications have compositional trace semantics or termination semantics, respectively.

**Related Work.** The above-mentioned *cool* GSOS rules of Bloom [7] and van Glabbeek [39] guarantee compositionality w.r.t. various flavours of weak bisimilarity; they motivate the *cool stateful SOS* format we introduce here. In a similar vein, Tsampas et al. [36] present abstract compositionality criteria for weak bisimilarity in the context of mathematical operational semantics [37]. Weak bisimilarity is still rather finer than the main semantics of interest for the present work (trace semantics and termination semantics), as it only abstracts away from steps that do not modify the state, such as `skip`.

Abou-Saleh and Pattinson [1, 2] consider abstract GSOS specifications for while-languages and construct semantics in Kleisli categories, working at a somewhat higher level of generality than we do here, in particular parametrizing over notions of side-effect. Roughly speaking, the coarsest of their semantics amounts to a *steps-until-termination* semantics that counts but does not enumerate intermediate states, and thus is coarser than trace semantics but finer than termination semantics. They propose an abstract *condition on cones* [1, Sec. 4.4] that guarantees compositionality for steps-until-termination semantics. This condition is hard to verify in concrete instances but ensured by *evaluation-in-context* rule formats [2] that correspond roughly to our *cool* stateful SOS format, for which we show compositionality even

w.r.t. termination semantics (a goal explicitly mentioned by Abou-Saleh and Pattinson [2, Section 6]). Our *streamlined* stateful SOS format, which guarantees compositionality of trace semantics, appears to be more permissive than evaluation-in-context.

Bloom and Vandraager [9] and Mousavi et al. [24] propose further SOS-style formats for computations with data and prove compositionality results for semantic equivalences resembling our resumption semantics. We note that these results require fairly tedious proofs; this again highlights the advantage of the categorical approach where they come entirely for free (see Theorem 4.6). The *Sfisl* format [24] is shown to make trace semantics compositional, but in contrast to our streamlined format it is not expressive enough to cover a fully fledged while-language. Termination semantics is not considered in either of these works.

## 2 Preliminaries

We assume that readers are familiar with basic notions from category theory such as functors, natural transformations, and monads. In the following we briefly recall some terminology concerning algebras and coalgebras. Throughout, **Set** denotes the category of sets and functions. We write $1 = \{*\}$ for the terminal object. For a pair $X_1, X_2$ of objects we write $X_1 \times X_2$ for the product with the projections $\mathsf{fst}\colon X_1 \times X_2 \to X_1$ and $\mathsf{snd}\colon X_1 \times X_2 \to X_2$. For a pair of morphisms $f_i\colon Y \to X_i$, $i = 1, 2$, we let $\langle f_1, f_2\rangle\colon Y \to X_1 \times X_2$ denote the unique induced morphism. The *canonical strength* of an endofunctor $F\colon \mathbf{Set} \to \mathbf{Set}$ is the natural transformation with components $\mathsf{st}_{X,Y}\colon X \times FY \to F(X \times Y)$ defined by $\mathsf{st}_{X,Y}(x, p) = F(\lambda y.\,(x, y))(p)$. We usually drop the subscripts $X$ and $Y$.

**Algebras.** Given an endofunctor $F$ on a category $\mathcal{C}$, an *$F$-algebra* is a pair $(A, \alpha)$ of an object $A$ (the *carrier* of the algebra) and a morphism $\alpha\colon FA \to A$ (its *structure*). A *homomorphism* from an $F$-algebra $(A, \alpha)$ to an $F$-algebra $(B, \beta)$ is a morphism $h\colon A \to B$ of $\mathcal{C}$ such that $h \cdot \alpha = \beta \cdot Fh$. Algebras for $F$ and their homomorphims form a category $\mathsf{Alg}\,F$, and an *initial* $F$-algebra is simply an initial object in that category. If it exists, we denote the initial $F$-algebra by $\mu F$ and its structure by $\iota\colon F(\mu F) \to \mu F$.

A common example of functor algebras are algebras over a signature. An *algebraic signature* consists of a set $\Sigma$ of operation symbols together with a map $\mathsf{ar}\colon \Sigma \to \mathbb{N}$ associating to every operation symbol $\mathsf{f}$ its *arity* $\mathsf{ar}(\mathsf{f})$. Symbols of arity 0 are called *constants*. Every signature $\Sigma$ induces the polynomial functor $\coprod_{\mathsf{f}\in\Sigma}(-)^{\mathsf{ar}(\mathsf{f})}$ on **Set**, which we denote by the same letter $\Sigma$. An algebra for the functor $\Sigma$ then is precisely an algebra for the signature $\Sigma$, i.e. a set $A$ equipped with an operation $\mathsf{f}_A\colon A^n \to A$ for every $n$-ary operation symbol $\mathsf{f} \in \Sigma$. Homomorphisms between $\Sigma$-algebras are maps respecting the algebraic structure.

Given a set $X$ of variables, we write $\Sigma^\star X$ for the $\Sigma$-algebra of terms generated by $\Sigma$ with variables from $X$. It is the *free $\Sigma$-algebra* on $X$, that is, every map $f\colon X \to A$ into the carrier of a $\Sigma$-algebra $(A, \alpha)$ uniquely extends to a homomorphism $\bar{f}\colon \Sigma^\star X \to A$. In particular, the free algebra on the empty set is the initial algebra $\mu\Sigma$; it is formed by all *closed terms* of the signature. As shown by Barr [4], the formation of free algebras extends to a monad $\Sigma^\star\colon \mathbf{Set} \to \mathbf{Set}$, the *free monad* on $\Sigma$. For every $\Sigma$-algebra $(A, \alpha)$ we obtain an Eilenberg-Moore algebra $\widehat{\alpha}\colon \Sigma^\star A \to A$ as the free extension of $\mathsf{id}_A$. This is the map evaluating terms over $A$ in the algebra.

**Coalgebras.** A *coalgebra* for an endofunctor $F$ on $\mathcal{C}$ is a pair $(C, \gamma)$ of an object $C$ (the *carrier*) and a morphism $\gamma\colon C \to FC$ (its *structure*). A *homomorphism* from an $F$-coalgebra $(C, \gamma)$ to an $F$-coalgebra $(D, \delta)$ is a morphism $h\colon C \to D$ such that $Fh \cdot \gamma = \delta \cdot h$. Coalgebras

for $F$ and their homomorphisms form a category $\mathsf{Coalg}\,F$, and a *final* coalgebra is a final object in that category. If it exists, we denote the final $F$-coalgebra by $\nu F$ and its structure by $\tau\colon \nu F \to F(\nu F)$, and we write $\gamma^\sharp\colon (C,\gamma) \to (\nu F,\tau)$ for the unique homomorphism.

▶ **Example 2.1.**

1. Fix a set $S$. The set functor $BX = S \times (X+1)$ has a final coalgebra carried by $\nu B = S^+ + S^\omega$, the set of all non-empty possibly terminating $S$-streams. Its coalgebra structure $S^+{+}S^\omega \to S \times (S^+ + S^\omega + 1)$ sends a stream $sw$ (where $s \in S$ and $w \in S^*{+}S^\omega$) to $(s,w)$ if $w \in S^+ + S^\omega$ and to $(s,*)$ if $w$ is empty.
2. Similarly, for the set functor $TX = (BX)^S = (S \times (X+1))^S$, the terminal coalgebra is carried by the set of possibly infinite $S$-ary trees (i.e. every node is either a leaf or has an $S$-indexed set of children) that have more than one node and where every edge is labelled by an element of $S$. The coalgebra structure $\nu T \to (S \times (\nu T + 1))^S$ sends a tree $t$ to the map $s \mapsto (s',t')$ where $s'$ is the label of the edge from the root to its $s$-th child, and $t'$ is the subtree rooted at that child if it has more than one node, or $*$ otherwise.

## 3 Stateful SOS Specifications

We start off with an observation on the standard operational semantics for sequential composition in imperative languages (see e.g. Plotkin [32]), given by the following rules:

$$\texttt{seq1}\,\frac{s,p \downarrow s'}{s,(p\,\texttt{;}\;q) \to s',q} \qquad\qquad \texttt{seq2}\,\frac{s,p \to s',p'}{s,(p\,\texttt{;}\;q) \to s',(p'\,\texttt{;}\;q)} \tag{3.1}$$

Rule `seq1` asserts that if a program $p$, on input (state) $s$, terminates and produces a new state $s'$, then the program $p\texttt{;}\;q$, on input state $s$, evolves to program $q$ and produces the new state $s'$. The other case is captured by rule `seq2`, which asserts that if $p$, on input $s$, transitions to $p'$ and produces $s'$, then $p\texttt{;}\;q$, on input $s$, transitions to $p'\texttt{;}\;q$ and produces $s'$. Note that for both rules, the *input* $s$ is the same in the premiss and in the conclusion. Consequently, to decide how $p\texttt{;}\;q$ transitions from $s$ in the next step, we need to know only how $p$ behaves on $s$, which we can regard as the input of the entire rule. This allows us to give a concise categorical formulation of the rules `seq1` and `seq2` in terms of a natural transformation $S \times (X \times S \times (X+1))^2 \to (S \times \Sigma^\star X + 1)$ where $\Sigma$ is a signature containing the binary operation symbol '`;`'. The transformation is defined by

$$(s,(x,s',*),(y,_,_)) \mapsto (s',y) \qquad \text{and} \qquad (s,(x,s',x'),(y,_,_)) \mapsto (s',(x'\,\texttt{;}\;y)).$$

Compare the above with the interpretation obtained by instantiating the GSOS principle [38] to stateful computations in the standard manner [37]. The interpretation of '`;`' is then given as a natural transformation $(X \times (S \times (X+1))^S)^2 \to (S \times (\Sigma^\star X + 1))^S$ whose uncurried form $S \times (X \times (S \times (X+1))^S)^2 \to S \times (\Sigma^\star X + 1)$ is defined by

$$(s,(x,f),(y,_)) \mapsto \begin{cases} (s',y) & \text{if } f(s) = (s',*), \\ (s',(x'\,\texttt{;}\;y)) & \text{if } f(s) = (s',x'). \end{cases}$$

In this setting, the semantics of $p\texttt{;}\;q$ receives the next-step behaviours of $p$, $q$ as state transformers, and can in principle probe these state transformers on arbitrary states (of course, for '`;`', this does not actually happen). By contrast, our rule format, the stateful SOS format formally introduced next, embodies the restriction that the behaviour of a complex term on an input state $s$ is predicated only on the behaviour of its subterms on $s$. It is this trade-off in expressiveness that buys our compositionality results for stateful SOS specifications.

**The Stateful SOS Rule Format.** We proceed to underpin the intuition given above with formal definitions. We fix a countably infinite set $\mathcal{V} = \{x_1, x_2, \dots\} \cup \{y_1, y_2, \dots\}$ of (*meta-*)*variables* and a countable set $S$ of *states*; in typical applications the elements of $S$ are variable stores. Moreover, we fix an algebraic signature $\Sigma$, equivalently a polynomial functor also denoted $\Sigma$ (cf. Section 2). We think of the operations in $\Sigma$ as program constructs, and correspondingly, *programs* are closed $\Sigma$-terms, i.e. terms formed using only the operations in $\Sigma$, with constants in $\Sigma$ forming the base case.

▶ **Definition 3.1** (Literals). A *progressing* $\Sigma$*-literal* is an expression $s, p \to s', q$ with $p, q \in \Sigma^\star\mathcal{V}$ and $s, s' \in S$. We say that $s$ is the *input*, $p$ is the *source*, $s'$ is the *output* and $q$ is the *target* of the literal. A *terminating* $\Sigma$*-literal* is an expression $s, p \downarrow s'$ with $s, s' \in S$ and $p \in \Sigma^\star\mathcal{V}$. In this case, $s$ is the input, $p$ is the source and $s'$ is the output of the literal. A $\Sigma$*-literal* (without further qualification) is either a progressing or a terminating $\Sigma$-literal.

Our rule format shares some similarities with *stream GSOS* [19, Def. 37].

▶ **Definition 3.2** (Rules). A *stateful SOS rule* for an $n$-ary operator $\mathsf{f} \in \Sigma$ is an expression

$$\frac{l_1 \quad \dots \quad l_n}{L} \tag{3.2}$$

(or, in inline notation, $l_1\ \dots\ l_n/L$) where $l_1, \dots, l_n$ (the *premisses* of the rule) and $L$ (the *conclusion* of the rule) are $\Sigma$-literals that have the same input $s \in S$, the *input* of the rule, and satisfy the following conditions:

1. The source of the premiss $l_j$ is the variable $x_j$, and the target is $y_j$ if $l_j$ is progressing.
2. The source of the conclusion $L$ is the term $\mathsf{f}(x_1, \dots, x_n)$. Moreover, if $L$ is progressing, the variables of its target term appear either as the source or the target of some premiss.

The rule is *progressing* if $L$ is progressing, and otherwise the rule is *terminating*. The *trigger* of the rule is the tuple formed by its input $s$ together with the sequence of pairs $\overrightarrow{(s', \mathsf{c})} = (s'_1, \mathsf{c}_1), \dots, (s'_n, \mathsf{c}_n)$, where $s'_j$ is the output of $l_j$ and $\mathsf{c}_j \in \{\mathtt{pr}, \mathtt{te}\}$ indicates whether $l_j$ is progressing ($c_j = \mathtt{pr}$) or terminating ($c_j = \mathtt{te}$).

▶ **Definition 3.3.** A *stateful SOS specification* is a set of stateful SOS rules such that for each $n$-ary operator $\mathsf{f}$, each $s \in S$ and each sequence $\overrightarrow{(s', \mathsf{c})} = (s'_1, \mathsf{c}_1), \dots, (s'_n, \mathsf{c}_n)$ where $s'_j \in S$ and $\mathsf{c}_j \in \{\mathtt{pr}, \mathtt{te}\}$, there is exactly one rule for $\mathsf{f}$ with trigger $(s, \overrightarrow{(s', \mathsf{c})})$.

▶ **Notation 3.4.** By writing

$$\frac{l_1 \quad \dots \quad l_{j-1} \quad l_{j+1} \quad \dots \quad l_n}{L}$$

we mean the set of all stateful SOS rules of the form $l_1\ \dots\ l_n/L$ (with the missing premiss $l_j$ filled in in any way possible). This captures the situation where the behaviour of the source $\mathsf{f}(x_1, \dots, x_n)$ of $L$ does not depend on the behaviour of $x_j$, given $l_1, \dots, l_{j-1}, l_{j+1}, \dots, l_n$.

▶ **Remark 3.5.** The use of fixed enumerated variables $x_1, x_2, \dots$ and $y_1, y_2, \dots$ simplifies abstract reasoning about stateful SOS (e.g. Theorem 3.9 below). In examples, we use arbitrary variable names such as $p, q, x, y$, and we typically write rules using rule schemes, using hopefully self-explanatory notation. For instance, rule `seq1` in Figure 1 (discussed in detail in Example 3.6) is to be understood as the set $\{\ s, p \downarrow s'\ /\ s, (p;\ q) \to s', q \mid s, s' \in S\}$ of stateful SOS rules, with variables $p, q$, and rule `while1` as the set $\{\ /s, \mathtt{while}\ e\ p \downarrow s \mid s \in S,\ [e]_s = 0\}$ (with premiss omitted as per Notation 3.4). Note the side condition $[e]_s = 0$ (expression $e$ evaluates to 0 in state $s$) of `while1`; the rule schemes and their side conditions need to be

$$\texttt{skip}\,\frac{}{s, \texttt{skip} \downarrow s} \qquad \texttt{asn}\,\frac{}{s, (x := e) \downarrow s_{[x \leftarrow [e]_s]}}$$

$$\texttt{while1}\,\frac{}{s, \texttt{while}\ e\ p \downarrow s}\,[e]_s = 0 \qquad \texttt{while2}\,\frac{}{s, \texttt{while}\ e\ p \to s, (p\texttt{;}\ \texttt{while}\ e\ p)}\,[e]_s \neq 0$$

$$\texttt{seq1}\,\frac{s, p \downarrow s'}{s, (p\texttt{;}\ q) \to s', q} \qquad \texttt{seq2}\,\frac{s, p \to s', p'}{s, (p\texttt{;}\ q) \to s', (p'\texttt{;}\ q)}$$

**Figure 1** Operational semantics of *While*.

set up in such a way that they actually obey the restrictions in Definition 3.3. For example, in the case of `while1` and `while2`, this is ensured by the respective side conditions ($[e]_s = 0$ and $[e]_s \neq 0$) being exhaustive and mutually exclusive.

▶ **Example 3.6.** We will use a prototypical imperative language, *While*, as a running example. Fix a countably infinite set $\mathcal{A}$ of program variables; then, the set $S$ of *stores* consists of all maps $s\colon \mathcal{A} \to \mathbb{N}$ whose *support* $\{x \in \mathcal{A} \mid s(x) \neq 0\}$ is finite. We denote by $s_{[x \leftarrow v]}$ the result of changing the value of variable $x$ to $v$ in a store $s$. Moreover, we assume a set $E$ of expressions that include the arithmetic operations $+, -, *$, constants $n \in \mathbb{N}$ and variables $x \in \mathcal{A}$. We write $[e]_s$ for the evaluation of expression $e$ under store $s$ (in the literature, evaluation is often defined stepwise by induction on the structure of the expression [32]; since this process does not affect the program state, we instead assume a denotational semantics for simplicity). The syntax of *While* is given by the grammar

$$\langle \text{prog} \rangle \ ::= \ \texttt{skip} \mid x := e \mid \langle \text{prog} \rangle\texttt{;}\ \langle \text{prog} \rangle \mid \texttt{while}\ e\ \langle \text{prog} \rangle \qquad (x \in \mathcal{A},\, e \in E),$$

which in terms of algebraic operations means that the signature $\Sigma$ includes constants `skip` and $x := e$ for all $x \in \mathcal{A}$, $e \in E$, a binary operation `;` and a unary operation `while` $e$ for each $e \in E$. The corresponding polynomial functor is

$$\Sigma X = 1 + \mathcal{A} \times E + X \times X + E \times X.$$

The operational semantics of *While* in the form of a stateful SOS specification is shown in Figure 1, using rule schemes as per Remark 3.5.

As indicated by the discussion at the beginning of this section, stateful SOS specifications can be represented as natural transformations:

▶ **Definition 3.7.** A *stateful SOS law* is a natural transformation

$$\delta_X \colon S \times \Sigma(X \times S \times (X + 1)) \to S \times (\Sigma^\star X + 1) \qquad (X \in \mathbf{Set}).$$

▶ **Remark 3.8.**

1. Every stateful SOS specification $\mathcal{L}$ yields a stateful SOS law

$$\delta_X = [\delta_X^{\mathsf{f}}]_{\mathsf{f} \in \Sigma} \colon S \times \Sigma(X \times S \times (X + 1)) \to S \times (\Sigma^\star X + 1) \qquad (X \in \mathbf{Set})$$

by distributing $S \times (-)$ over $\Sigma(X \times S \times (X + 1))$ and copairing the maps

$$\delta_X^{\mathsf{f}} \colon S \times (X \times S \times (X + 1))^{\mathsf{ar}(\mathsf{f})} \to S \times (\Sigma^\star X + 1) \qquad (\mathsf{f} \in \Sigma) \tag{3.3}$$

defined as follows. Given $(s,((v_1,s'_1,w_1),\dots,(v_n,s'_n,w_n))) \in S\times(X\times S\times(X+1))^n$ with $n=\mathsf{ar}(f)$, let $l_1\ \dots\ l_n/L$ be the unique rule in $\mathcal{L}$ with source $\mathsf{f}$ and trigger $(s,((s'_1,c_1),\dots,(s'_n,c_n)))$ where $c_j=\mathtt{pr}$ if $w_j\in X$ and $c_j=\mathtt{te}$ if $w_j=*$. Let $s'$ be the output of $L$. Then $\delta^{\mathsf{f}}_X(s,((v_1,s'_1,w_1),\dots,(v_n,s'_n,w_n)))$ is $(s',*)$ if the rule is terminating, and otherwise $(s',t')$ where $t'\in\Sigma^\star X$ is the term obtained from the target $t\in\Sigma^\star\mathcal{V}$ of $L$ by substituting $x_j$ by $v_j$ and $y_j$ by $w_j$ (the latter whenever $c_j=\mathtt{pr}$).

2. Conversely, every stateful SOS law $\delta$ yields a stateful SOS specification $\mathcal{L}$ whose rules are defined as follows. For every $n$-ary operation symbol $\mathsf{f}\in\Sigma$, $s,s'_1,\dots,s'_n\in S$ and $W\subseteq\{1,\dots,n\}$, let $(s',t)$ be the value of $\delta^{\mathsf{f}}_{\mathcal{V}}$ on $(s,((x_1,s'_1,w_1),\dots,(x_n,s'_n,w_n)))$ where $w_j=y_j$ if $j\in W$ and $w_j=*$ otherwise. If $t\in\Sigma^\star\mathcal{V}$, then $\mathcal{L}$ contains the rule

$$\frac{(s,x_j\to s'_j,y_j)_{j\in W}\qquad (s,x_j\downarrow s'_j)_{j\in\{1,\dots,n\}\smallsetminus W}}{s,\mathsf{f}(x_1,\dots,x_n)\to s',t},$$

and if $t=*$, then $\mathcal{L}$ contains the rule

$$\frac{(s,x_j\to s'_j,y_j)_{j\in W}\qquad (s,x_j\downarrow s'_j)_{j\in\{1,\dots,n\}\smallsetminus W}}{s,\mathsf{f}(x_1,\dots,x_n)\downarrow s'}.$$

▶ **Theorem 3.9.** *There is a bijective correspondence between (1) stateful SOS specifications, (2) stateful SOS laws, and (3) families of maps of the form*

$$\big(r_{\mathsf{f},W}\colon S\times S^{\mathsf{ar}(\mathsf{f})}\to S\times\Sigma^\star(\mathsf{ar}(\mathsf{f})+W)+S\big)_{\mathsf{f}\in\Sigma,W\subseteq\mathsf{ar}(\mathsf{f})}.$$

*Here we identify the natural number* $\mathsf{ar}(\mathsf{f})$ *with the set* $\{1,\dots,\mathsf{ar}(\mathsf{f})\}$.

The correspondence between (1) and (2) is given by the translations of Remark 3.8, and the correspondence between (2) and (3) is shown using the Yoneda lemma.

## 4 Categorical Semantics and Compositionality

We proceed to develop a categorical treatment of stateful SOS along the lines of mathematical operational semantics in the style of Turi and Plotkin [38] and Bartels [5]. Furthermore, we shall define two semantic domains of interest, both coarser than the one initially obtained through Turi-Plotkin semantics, and show that the problem of whether a given stateful SOS specification is compositional is undecidable. We recall that if the denotational semantics of a programming language is given by a map $[\![-]\!]\colon\mu\Sigma\to D$ into a semantic domain $D$, then it is called *compositional* if the corresponding behavioural equivalence forms a congruence, that is, for every $n$-ary operator $\mathsf{f}\in\Sigma$ and programs $p_i,q_i\in\mu\Sigma$ $(i=1,\dots,n)$,

$$[\![p_i]\!]=[\![q_i]\!]\text{ for }i=1,\dots,n\qquad\text{implies}\qquad[\![\mathsf{f}(p_1,\dots,p_n)]\!]=[\![\mathsf{f}(q_1,\dots,q_n)]\!].$$

Compositionality asserts that subprograms of a program $p$ may be replaced with equivalent subprograms without affecting the semantics of $p$, and thus allows modular reasoning.

### 4.1 GSOS Laws

Turi and Plotkin's *mathematical operational semantics* [38] identifies sets of rules in structural operational semantics (SOS) with distributive laws of various types on a cartesian base category. We will work more specifically with distributive laws of free monads over cofree copointed functors on the base category **Set**, where the free monad is associated to a polynomial functor. Such distributive laws can equivalently be presented as follows.

▶ **Definition 4.1.** Given a polynomial functor $\Sigma$ and an endofunctor $T$ on **Set**, a *GSOS law of* $\Sigma$ *over* $T$ is a natural transformation $\varrho\colon \Sigma(\mathrm{Id} \times T) \Longrightarrow T\Sigma^\star$.

We shall see below that stateful SOS laws determine GSOS laws. The interested reader may find further examples of GSOS laws in the literature [37, 5, 19]. Roughly speaking, the input of $\varrho$ is a (program) operation applied to pairs each consisting of a meta-variable and its assumed next-step behaviour (encapsulated in $T$), and the output is a next-step behaviour reaching poststates given as programs with meta-variables.

Given a GSOS law $\varrho$, the initial $\Sigma$-algebra can be equipped with a unique $T$-coalgebra structure $\gamma\colon \mu\Sigma \to T(\mu\Sigma)$ such that the diagram

$$\begin{array}{ccccc} \Sigma(\mu\Sigma) & & \xrightarrow{\quad\iota\quad} & & \mu\Sigma \\ \downarrow{\scriptstyle \Sigma\langle \mathsf{id},\gamma\rangle} & & & & \downarrow{\scriptstyle\gamma} \\ \Sigma(\mu\Sigma \times T(\mu\Sigma)) & \xrightarrow{\varrho_{\mu\Sigma}} & T\Sigma^\star(\mu\Sigma) & \xrightarrow{T\hat{\iota}} & T(\mu\Sigma) \end{array} \tag{4.1}$$

commutes (see Section 2 for the notation). The coalgebra $(\mu\Sigma, \gamma)$ is called the *operational model* of $\varrho$. Dually, assuming the existence of a final coalgebra $\nu T$, there is a unique $\Sigma$-algebra structure $\alpha\colon \Sigma(\nu T) \to \nu T$ such that the following diagram commutes:

$$\begin{array}{ccccc} \Sigma(\nu T) & \xrightarrow{\Sigma\langle \mathsf{id},\tau\rangle} & \Sigma(\nu T \times T(\nu T)) & \xrightarrow{\varrho_{\nu T}} & T\Sigma^\star(\nu T) \\ \downarrow{\scriptstyle\alpha} & & & & \downarrow{\scriptstyle T\hat{\alpha}} \\ \nu T & & \xrightarrow{\quad\tau\quad} & & T(\nu T) \end{array} \tag{4.2}$$

The algebra $(\nu T, \alpha)$ is the *denotational model* of $\varrho$. A fundamental well-behavedness property of GSOS laws is that the unique $\Sigma$-algebra homomorphism $(\mu\Sigma, \iota) \to (\nu T, \alpha)$ and the unique $T$-coalgebra homomorphism $(\mu\Sigma, \gamma) \to (\nu T, \tau)$ coincide. We denote this morphism by

$$\mathsf{beh}_\varrho\colon \mu\Sigma \to \nu T, \tag{4.3}$$

and we think of it as assigning to programs their denotational behaviour. Compositionality of this semantics is immediate from the fact that $\mathsf{beh}_\varrho$ is a $\Sigma$-algebra homomorphism.

## 4.2 Semantic Domains for Stateful SOS

We proceed to introduce three denotational semantics of stateful SOS, in order of increasing abstraction: *resumption semantics*, in which the program essentially cedes control to the environment between any two program steps; *trace semantics*, where the environment may observe but not manipulate the state between program steps; and *termination semantics*, in which only the effect of executing the program end-to-end is observable.

▶ **Notation 4.2.** From now on, we instantiate the functor $T$ of Definition 4.1 to

$$TX = (S \times (X + 1))^S,$$

for a fixed set $S$ of states. Thus $T$ represents state transformers with possible non-termination.

**Resumption semantics.** Every stateful SOS law $\delta$ (see Definition 3.7) canonically induces a GSOS law

$$\hat{\delta}\colon \Sigma(\mathrm{Id} \times T) \Longrightarrow T\Sigma^\star.$$

This will guarantee compositionality for the most fine-grained of our semantics, which we shall refer to as *resumption semantics*, via established methods of mathematical operational semantics as recalled above. Details are as follows. The component $\hat{\delta}_X$ is obtained by currying the composite

$$\begin{aligned} S \times \Sigma(X \times TX) &\xrightarrow{\langle \mathsf{fst}, \mathsf{st} \rangle} S \times \Sigma(S \times (X \times TX)) \cong S \times \Sigma(X \times (S \times TX)) \\ &\xrightarrow{\mathsf{id} \times \Sigma(\mathsf{id} \times \mathsf{ev})} S \times \Sigma(X \times S \times (X+1)) \xrightarrow{\delta_X} S \times (\Sigma^\star X + 1), \end{aligned} \tag{4.4}$$

where $\mathsf{st}\colon S \times \Sigma(X \times TX) \to \Sigma(S \times (X \times TX))$ is the strength (cf. Section 2) and $\mathsf{ev}\colon S \times TX = S \times (S \times (X+1))^S \to S \times (X+1)$ denotes the evaluation map. Recall from Example 2.1 that the final coalgebra for $T$ is carried by the set of possibly infinite $S$-branching trees, with edges labelled in $S$. Using (4.1) we obtain the operational model $\gamma\colon \mu\Sigma \to T(\mu\Sigma)$ associated to $\hat{\delta}$. In terms of stateful SOS specifications, it can be described as follows.

▶ **Definition 4.3.** Given a stateful SOS specification $\mathcal{L}$, its *transition function* is the map

$$\gamma_0\colon S \times \mu\Sigma \to S \times (\mu\Sigma + 1)$$

inductively defined by

$$\gamma_0(s, \mathsf{f}(t_1, \ldots, t_n)) = m(\delta^{\mathsf{f}}_{\mu\Sigma}(s, (d_1, \ldots, d_n)))$$

where

$$d_j = (t_j, \gamma_0(s, t_j)) \qquad \text{and} \qquad m = \big(S \times (\Sigma^\star(\mu\Sigma) + 1) \xrightarrow{\mathsf{id} \times (\hat{\iota} + \mathsf{id})} S \times (\mu\Sigma + 1)\big),$$

using the term evaluation map $\hat{\iota}\colon \Sigma^\star(\mu\Sigma) \to \mu\Sigma$, and $\delta^{\mathsf{f}}_{\mu\Sigma}$ as in (3.3). Thus, $\gamma_0(s, p)$ performs the first computation step of program $p$ on input $s$ according to the specification $\mathcal{L}$. We write

$$s, p \to s', p' \qquad \text{and} \qquad s, p \downarrow s'$$

if $\gamma_0(s, p) = (s', p')$ and $\gamma_0(s, p) = (s', *)$, respectively.

▶ **Proposition 4.4.** *Let $\mathcal{L}$ be a stateful SOS specification with its associated transition function $\gamma_0$ and operational model $\gamma$. Then*

$$\gamma = \mathsf{curry}(\gamma_0)\colon \mu\Sigma \to (S \times (\mu\Sigma + 1))^S.$$

The proof makes use of an induction principle that combines *primitive recursion* (see e.g. [16, Prop. 2.4.7]) and *induction with parameters* (see e.g. [16, Exercise 2.5.5]).

▶ **Definition 4.5.** The *resumption semantics* of a stateful SOS specification $\mathcal{L}$ is given by

$$[-]_{\mathcal{L}} = \mathsf{beh}_{\hat{\delta}}\colon \mu\Sigma \to \nu T,$$

where $\delta$ is the stateful SOS law associated to $\mathcal{L}$, $\hat{\delta}$ is as per (4.4), and $\mathsf{beh}$ is defined in (4.3). Let $\sim_{\mathcal{L}}$ denote the corresponding behavioural equivalence, that is, $p \sim_{\mathcal{L}} q$ iff $[p]_{\mathcal{L}} = [q]_{\mathcal{L}}$ for a given pair $p, q \in \mu\Sigma$. We drop subscripts if $\mathcal{L}$ is clear from the context.

Note that since $T$ preserves weak pullbacks, $\sim_{\mathcal{L}}$ coincides with $T$-bisimilarity in the operational model $\gamma\colon \mu\Sigma \to T(\mu\Sigma)$ [34]. From the discussion in Section 4.1 we immediately get

▶ **Theorem 4.6.** *The resumption semantics of stateful SOS specifications is compositional.*

Resumption semantics is very fine-grained, essentially because it does not pass the output state of a computation step on as the input state of the next step; that is, resumption semantics assumes that the environment takes complete control in between steps. For instance, consider the *While* programs

$$t_1 = \big(\mathtt{x} \coloneqq 1;\ \mathtt{x} \coloneqq \mathtt{x}+1\big) \qquad \text{and} \qquad t_2 = \big(\mathtt{x} \coloneqq 1;\ \mathtt{x} \coloneqq \mathtt{x}*2\big).$$

The resumption semantics of these programs in each case consists in an $S$-branching tree of depth 2, in which the edge from the root to its $s$-th child is labelled $s[x \leftarrow 1]$ and the edges at the next level are correspondingly labelled according to the effect of the assignments $\mathtt{x} \coloneqq \mathtt{x}+1$ and $\mathtt{x} \coloneqq \mathtt{x}*2$, respectively. In particular, the semantics of the two programs differ – as intuitively expected under a resumption semantics, since the environment may manipulate the value of $x$ in between the two assignments. To obtain a more coarse-grained notion of process equivalence, we have to quotient the semantic domain $\nu T$ further.

**Trace Semantics.** Consider the set functor $B$ given by

$$BX = S \times (X+1);$$

thus $TX = (BX)^S$. Recall from Example 2.1 that the final coalgebra $\nu B$ is carried by the set $S^+ + S^\omega$ of possibly terminating $S$-streams. The set $(\nu B)^S$ serves as the semantic domain for *trace semantics* for imperative programs [25, 27, 28], which associates to a program the possibly terminating sequence of states it computes from a given initial state. In order to formally introduce trace semantics in our setting, we proceed to construct a quotient map $\nu T \twoheadrightarrow (\nu B)^S$ by coinduction. To this end, we define the functor $\overline{(-)}\colon \mathsf{Coalg}\,T \to \mathsf{Coalg}\,B$, which maps a $T$-coalgebra $(C,\zeta)$ to the $B$-coalgebra

$$\bar\zeta = S \times C \xrightarrow{\mathsf{id}\times\zeta} S \times (BC)^S \xrightarrow{\mathsf{ev}} BC = S \times (C+1) \xrightarrow{\langle \mathsf{fst},\mathsf{st}\rangle} S \times (S \times C + 1) = B(S \times C),$$

where $\mathsf{st}\colon S \times (C+1) \to S \times C+1$ is the strength of the functor $(-)+1$, given by $(s,c) \mapsto (s,c)$ and $(s,*) \mapsto *$. Intuitively, while $\zeta^\sharp\colon C \to \nu T$ (see Section 2 for the notation) maps a coalgebra state of $C$ to its tree of state transformers, $\overline{\zeta}^\sharp(s,x) \in \nu B$ executes all these state transformers without interruption, beginning at $s$ and feeding the output state of each previous step to the next step, and outputs the intermediate states reached in each step. Applying $\overline{(-)}$ to the final coalgebra $(\nu T,\tau)$, we obtain a $B$-coalgebra $(S \times \nu T, \overline{\tau})$, and currying the unique coalgebra homomorphism $\overline{\tau}^\sharp\colon S \times \nu T \to \nu B$ yields the desired quotient map

$$\mathsf{trc} = \mathsf{curry}(\overline{\tau}^\sharp)\colon \nu T \twoheadrightarrow (\nu B)^S. \tag{4.5}$$

▶ **Proposition 4.7.** *The map* trc *is surjective.*

▶ **Definition 4.8.** The *trace semantics* of a stateful SOS specification $\mathcal{L}$ is given by

$$[\![-]\!]_{\mathcal{L}} = (\mu\Sigma \xrightarrow{(\!|-|\!)_{\mathcal{L}}} \nu T \xrightarrow{\mathsf{trc}} (\nu B)^S).$$

Let $\simeq_{\mathcal{L}}$ denote the corresponding behavioural equivalence, that is, $p \simeq_{\mathcal{L}} q$ iff $[\![p]\!]_{\mathcal{L}} = [\![q]\!]_{\mathcal{L}}$, for $p,q \in \mu\Sigma$. We drop subscripts if $\mathcal{L}$ is clear from the context.

▶ **Remark 4.9.** Equivalently, $[\![-]\!]_{\mathcal{L}}$ is the curried form of the unique $B$-coalgebra homomorphism from $(S \times \mu\Sigma, \bar\gamma)$ to $\nu B$ (recall that $(\mu\Sigma,\gamma)$ is the operational model of $\mathcal{L}$). Since

$$\bar\gamma = \big(S \times \mu\Sigma \xrightarrow{\gamma_0} S \times (\mu\Sigma+1) \xrightarrow{\langle \mathsf{fst},\mathsf{st}\rangle} S \times (S \times \mu\Sigma + 1) = B(S \times \mu\Sigma)\big)$$

by definition of $\bar{\gamma}$ and Proposition 4.4, we see that for every $p \in \mu\Sigma$ and $s \in S$, the possibly infinite stream $[\![p]\!]_{\mathcal{L}}(s) = s_1 s_2 s_3 \cdots$ is the sequence of states computed by the program $p$ on input state $s$, cf. Definition 4.3:

$$s, p \to s_1, p_1 \to s_2, p_2 \to s_3, p_3 \to \cdots .$$

Hence trace equivalence $p \simeq q$ holds iff for each input state $s$, programs $p$ and $q$ produce the same sequence of states.

The following example demonstrates that trace semantics is generally not compositional:

▶ **Example 4.10.** We extend *While* by adding a unary operator $\lfloor \cdot \rfloor$ with

$$\frac{s, p \to s', p'}{s, \lfloor p \rfloor \to \emptyset, \lfloor p' \rfloor} \qquad \frac{s, p \downarrow s'}{s, \lfloor p \rfloor \downarrow s'}$$

where $\emptyset$ denotes the store with all variables set to 0. For $t_1 = \big(\mathtt{x} \coloneqq 1;\ \mathtt{x} \coloneqq \mathtt{x} + 1\big)$ and $t_2 = \big(\mathtt{x} \coloneqq 1;\ \mathtt{x} \coloneqq \mathtt{x} * 2\big)$, we have that $t_1 \simeq t_2$ but $\lfloor t_1 \rfloor \not\simeq \lfloor t_2 \rfloor$ (since in $\lfloor t_1 \rfloor$ and $\lfloor t_2 \rfloor$, the store is erased after the first assignment).

**Termination Semantics.** As the coarsest of our semantic domains, we shall use the set $(S + \{\bot\})^S \cong (S+1)^S$ of state transformers on $S$ with possible non-termination featuring pervasively in the denotational semantics of imperative programming (e.g. [33, 31, 30]). In comparison to $(\nu B)^S$, this domain abstracts from the intermediate steps of the computation. The essence of this abstraction is captured by the map

$$\mathsf{fn}\colon \nu B \to S + 1 \qquad \text{defined by} \qquad \mathsf{fn}(x) = \begin{cases} s & \text{if } x \text{ is finite, with last state } s, \\ \bot & \text{otherwise.} \end{cases}$$

▶ **Definition 4.11.** The *termination semantics* of a stateful SOS specification $\mathcal{L}$ is given by

$$[\![\![-]\!]\!]_{\mathcal{L}} = (\mu\Sigma \xrightarrow{[\![-]\!]_{\mathcal{L}}} (\nu B)^S \xrightarrow{\mathsf{fn}^S} (S+1)^S).$$

Let $\approx_{\mathcal{L}}$ denote the corresponding behavioural equivalence, that is, $p \approx_{\mathcal{L}} q$ iff $[\![\![p]\!]\!]_{\mathcal{L}} = [\![\![q]\!]\!]_{\mathcal{L}}$ for $p, q \in \mu\Sigma$. We drop subscripts if $\mathcal{L}$ is clear from the context.

Thus $p \approx q$ iff for each initial state $s$, if $p$ eventually terminates with final state $s'$ then $q$ eventually terminates with final state $s'$ and vice-versa. Termination semantics is generally not compositional: the programs $t_1$ and $t_2$ of Example 4.10 satisfy $t_1 \approx t_2$ but $\lfloor t_1 \rfloor \not\approx \lfloor t_2 \rfloor$.

The maps introduced in this section are summarized in the following commutative diagram:

$$\begin{array}{ccccc} & & \mu\Sigma & & \\ & \swarrow^{[-]_{\mathcal{L}}} & \downarrow^{[\![-]\!]_{\mathcal{L}}} & \searrow^{[\![\![-]\!]\!]_{\mathcal{L}}} & \\ \nu T & \xrightarrow{\mathsf{trc}} & (\nu B)^S & \xrightarrow{\mathsf{fn}^S} & (S+1)^S \end{array} \tag{4.6}$$

## 4.3 Compositionality is Undecidable

We have seen that in contrast to resumption semantics, both trace and termination semantics generally fail to be compositional. As it turns out, reasoning about compositionality in these two cases is a very complex, viz. undecidable, task.

To make the ensuing decision problems precise, we fix suitable encodings of states and terms as finite strings and regard a stateful SOS specification $\mathcal{L}$ as a total function that assigns to a given operation symbol, input state and list of premisses the target of the conclusion and output state of the respective rule. From a computational point of view, a minimum requirement on every reasonable specification $\mathcal{L}$ is that it admits some finite representation. Hence, for simplicity, we assume in the following theorem that specifications are primitive recursive functions. For instance, this is clearly the case for the *While* language.

▶ **Theorem 4.12.** *It is undecidable whether the trace semantics (or termination semantics, respectively) induced by a primitive recursive stateful SOS specification is compositional.*

**Proof sketch.** The halting problem reduces to the compositionality problem. The idea is to take programs akin to $t_1$ and $t_2$ in Example 4.10 and precompose them with the simulation of a given Turing machine. This can be specified in stateful SOS. The failure of compositionality described in Example 4.10 then occurs if, and only if, the simulated machine halts. ◀

In view of the fact that there is no sound and complete decision procedure for compositionality w.r.t. $\simeq$ and $\approx$, we instead move on to identify easily checked *sound* syntactic criteria that, although necessarily incomplete, are sufficiently broad.

## 5 Cooling the Stateful SOS Format

We now introduce two sets of restrictions on the stateful SOS rule format, called *streamlined stateful SOS* and *cool stateful SOS*, that guarantee trace and termination semantics, respectively, to be compositional. Our approach is inspired by the work of Bloom [7] and van Glabbeek [39] on the *cool* congruence formats for weak bisimilarity for GSOS specifications. The following definition will help describe the restricted formats. We make pervasive use of the abbreviations from Notation 3.4, and we will additionally employ $s, p \to s', *$ as an alternative notation for a terminating literal $s, p \downarrow s'$.

▶ **Definition 5.1.** Let $\mathcal{L}$ be a stateful SOS specification.

1. An $n$-ary operator $\mathsf{f}$ is *passive* if all rules for $\mathsf{f}$ are of the form

$$\frac{}{s, \mathsf{f}(x_1, \ldots, x_n) \to s', t} \qquad \text{where } t \in \Sigma^\star(\{x_1, \ldots, x_n\}) \text{ or } t = *.$$

   In other words, the one-step behaviour of $\mathsf{f}(x_1, \ldots, x_n)$ does not depend on the one-step behaviour of any of its subterms. In particular, every constant is passive. An *active* operator is one which is not passive.
2. A progressing rule for an $n$-ary operator $\mathsf{f}$ is *receiving at position* $j \in \{1, \ldots, n\}$ if its $j$-th premiss $s, x_j \to s', y_j$ is progressing and the variable $y_j$ appears in the target of the conclusion. We say that the rule is *receiving* if it is receiving at some position $j$.

### 5.1 Streamlined Stateful SOS

As indicated above, the streamlined Stateful SOS format, introduced next, will guarantee compositionality of trace semantics.

▶ **Definition 5.2.** A stateful SOS specification is *streamlined* if for every active operator $\mathsf{f}$ of arity $n$ there exists $j \in \{1, \ldots, n\}$ (the *receiving position* of $\mathsf{f}$) such that the following holds:

1. All receiving rules for f are of the form

$$\frac{s, x_j \to s', y_j}{s, \mathsf{f}(x_1, \ldots, x_n) \to s', t} \qquad \text{where } t = \mathsf{f}(x_1, \ldots, x_n)[y_j/x_j] \text{ or } t = y_j;$$

here, $[u/x]$ denotes substitution of the variable $x$ by the term $u$.

2. All non-receiving rules for f are of the form

$$\frac{l_1 \quad l_2 \quad \cdots \quad l_n}{s, \mathsf{f}(x_1, \ldots, x_n) \to s', t} \qquad \text{where } t \in \Sigma^\star(\{x_1, \ldots, x_n\} \setminus \{x_j\}) \text{ or } t = *.$$

Note that in a stateful SOS specification, receiving rules for an active operator f are receiving *only* in the receiving position of f. What Definition 5.2 boils down to is that an active operator can only progress its subterm at the receiving position $j$, leaving everything else unchanged and making sure that the output state in the $j$-th premiss is correctly propagated, and discards the $j$-th subterm once it terminates.

▶ **Example 5.3.** The *While* language (cf. Figure 1) is streamlined. The only active operator is sequential composition $p\mathtt{;}\ q$. Its progressing rules are receiving in the left position, and upon termination the left subterm is discarded.

Further examples are discussed after Corollary 5.5.

▶ **Theorem 5.4.** *Trace semantics is compositional for streamlined stateful SOS specifications.*

**Proof sketch.** For $p, q \in \mu\Sigma$ and $k \in \mathbb{N}$ we put $p \simeq_k q$ if the programs $p$ and $q$ are $k$-step trace equivalent, that is, for every $s \in S$ the streams $[\![p]\!](s)$ and $[\![q]\!](s)$ have the same prefix of length at most $k$. By induction on $k$ one proves $\simeq_k$ to be a congruence, using a judicious strengthening of the inductive claim for receiving positions of active operators. This implies that $\simeq$ is a congruence, whence trace semantics is compositional. ◀

From Theorem 5.4 we can deduce a slightly stronger statement. In what follows, the *kernel* of a map $e\colon X \to Y$ is the equivalence relation on $X$ relating $x, x'$ iff $e(x) = e(x')$.

▶ **Corollary 5.5.** *For every streamlined stateful SOS specification, the kernel of the map* $\mathsf{trc}\colon \nu T \twoheadrightarrow (\nu B)^S$ *is a congruence w.r.t. the canonical* $\Sigma$*-algebra structure on* $\nu T$ *as per* (4.2).

We next look at examples of streamlined specifications but also at a few pathological cases where compositionality breaks.

▶ **Example 5.6.** Streamlined specifications allow for complex control flow over programs, including *signal* or *interrupt handling.* For instance, we can extend *While* by a distinguished variable i serving as an interrupt flag and modify the rules of sequential composition to

$$\frac{s, p \downarrow s'}{s, (p\mathtt{;}\ q) \to s', q} \qquad \frac{s, p \to s', p'}{s, (p\mathtt{;}\ q) \to s', (p'\mathtt{;}\ q)}\,[\mathtt{i}]_s = 0$$

$$\frac{s, p \to s', p'}{s, (p\mathtt{;}\ q) \to s', q}\,[\mathtt{i}]_s \neq 0 \wedge P(s') \qquad \frac{s, p \to s', p'}{s, (p\mathtt{;}\ q) \to s', (p'\mathtt{;}\ q)}\,[\mathtt{i}]_s \neq 0 \wedge \neg P(s')$$

where $P \subseteq S$. If flag i is enabled and predicate $P$ is true for the output $s'$ of $p$, then $p$ is terminated prematurely. This type of rules can also be used to implement *listeners* or *observers* in high-level programming languages [17].

▶ **Example 5.7.**

1. Recall the operator $\lfloor \cdot \rfloor$ from Example 4.10, which breaks compositionality for trace semantics. The operator is active, and its progressing rule is receiving but does not propagate the output state of its premiss, so the stateful SOS specification of *While* with $\lfloor \cdot \rfloor$ fails to be streamlined (as it must, by Theorem 5.4).
2. Consider the extension of *While* with a binary left-first interleaving operator $\triangleleft$ specified by the rules

$$\frac{s, p \to s', p'}{s, p \triangleleft q \to s', q \triangleleft p'} \qquad \frac{s, p \downarrow s'}{s, p \triangleleft q \to s', q}$$

Again, $\simeq$ is not a congruence: For $t_1 = (\mathtt{x} := 2\mathtt{;}\ \mathtt{x} := \mathtt{x} + 2)$ and $t_2 = (\mathtt{x} := 2\mathtt{;}\ \mathtt{x} := \mathtt{x} * 2)$, we have $t_1 \simeq t_2$ but $t_1 \triangleleft (\mathtt{x} := 0) \not\simeq t_2 \triangleleft (\mathtt{x} := 0)$. Indeed the left of the above rules is receiving but the target of its conclusion does not have one of the allowed forms.
3. Extend *While* with a step-by-step branching operator $\triangledown$ specified by

$$\frac{s, p \to s_1, p' \qquad s, q \to s_2, q'}{s, p \triangledown q \to s_1, p' \triangledown q} P(s) \qquad \frac{s, p \to s_1, p' \qquad s, q \to s_2, q'}{s, p \triangledown q \to s_2, p \triangledown q'} \neg P(s)$$

and termination in all other cases. If the predicate $P \subseteq S$ is, for example, $\mathtt{x} = 0$, then the same $t_1, t_2$ as in item 2 witness that $\simeq$ is not a congruence: We have $t_1 \simeq t_2$ but $t_1 \triangledown (\mathtt{x} := 0) \not\simeq t_2 \triangledown (\mathtt{x} := 0)$. In this case, the condition that is violated is the requirement that all rules for $\triangledown$ must be receiving in the same position.
4. Consider the operator $\lceil \cdot \rceil$ specified by

$$\frac{s, p \to s', p'}{s, \lceil p \rceil \to s', \lceil p' \rceil} \qquad \frac{s, p \downarrow s'}{s, \lceil p \rceil \to s', p}$$

Again, $t_1, t_2$ as in item 2 witness failure of congruence: $t_1 \simeq t_2$ but $\lceil t_1 \rceil \not\simeq \lceil t_2 \rceil$. Indeed, the second rule violates Definition 5.2 as $p$ terminates but is not discarded.

## 5.2 Cool stateful SOS

We now further restrict the streamlined format as follows:

▶ **Definition 5.8.** A stateful SOS specification is *cool* if for every active operator $\mathsf{f}$ there exists $j \in \{1, \dots, n\}$ (again called the *receiving position of* $\mathsf{f}$) such that the following holds:

1. All rules for $\mathsf{f}$ whose $j$-th premiss is progressing are of the form

$$\frac{s, x_j \to s', y_j}{s, \mathsf{f}(x_1, \dots, x_n) \to s', \mathsf{f}(x_1, \dots, x_n)[y_j / x_j]}$$

2. All rules for $\mathsf{f}$ whose $j$-th premiss is terminating are of the form

$$\frac{s, x_j \downarrow s'}{s, \mathsf{f}(x_1, \dots, x_n) \to s'', t} \qquad \text{where } t \in \Sigma^\star(\{x_1, \dots, x_n\} \smallsetminus \{x_j\}) \text{ or } t = *,$$

and moreover $s''$ and $t$ depend only on $s'$ but not on $s$.

A stateful SOS specification is *uncool* if it is not cool.

The cool format asserts that an active operator $\mathsf{f}$ runs its $j$-th subterm until termination and then discards it, proceeding to a state derivable from the terminating state of the subterm. In GSOS, rules of type 1 (without states) are known as *patience rules* [39].

▶ **Example 5.9.** The rules of the *While* language, which we have already observed to be streamlined (Example 5.3), are also cool.

Cool stateful SOS specifications are streamlined, and all of the negative examples from Section 5.1 apply here as well. Here is an example that separates the two concepts:

▶ **Example 5.10.** The sequential composition semantics with interrupts from Example 5.6 is uncool, as the third rule has a progressing premiss but is not of the form in Definition 5.8.1. Indeed, $\approx$ is not a congruence: For the predicate $\mathtt{x} = 42$ and the programs $t_1 = (\mathtt{x}{:=}42\mathtt{;}\ \mathtt{x}{:=}2)$ and $t_2 = (\mathtt{x} := 2)$, we have $t_1 \approx t_2$ but $t_1\mathtt{;}\ \mathtt{skip} \not\approx t_2\mathtt{;}\ \mathtt{skip}$.

As indicated above, coolness guarantees congruence for termination semantics:

▶ **Theorem 5.11.** *Termination semantics is compositional for cool stateful SOS specifications.*

**Proof sketch.** Suppose that $\mathsf{f} \in \Sigma$ is an $n$-ary operator and $p_m, q_m \in \mu\Sigma$ are programs with $p_m \approx q_m$ for $m = 1, \ldots, n$. By symmetry, it suffices to show the following for all $s, \overline{s} \in S$:

If $s, \mathsf{f}(p_1, \ldots, p_m)$ terminates in state $\overline{s}$, then $s, \mathsf{f}(q_1, \ldots, q_m)$ terminates in state $\overline{s}$.

The proof proceeds by an outer induction on the number of steps until termination of $s, \mathsf{f}(p_1, \ldots, p_m)$ and an inner induction on the structure of the programs. ◀

By Corollary 5.5 we know that for every cool (whence streamlined) specification the kernel of $\mathtt{trc}\colon \nu T \twoheadrightarrow (\nu B)^S$ forms a congruence. Since $\mathtt{trc}$ is surjective, this means precisely that there is a (unique) $\Sigma$-algebra structure on $(\nu B)^S$ for which $\mathtt{trc}$ is a $\Sigma$-algebra homomorphism.

▶ **Corollary 5.12.** *For every cool stateful SOS specification, the kernel of* $\mathsf{fn}^S\colon (\nu B)^S \twoheadrightarrow (S+1)^S$ *is a congruence w.r.t. the induced* $\Sigma$*-algebra structure on* $(\nu B)^S$.

## 6 Conclusions and Future Work

We have introduced the *stateful SOS* rule format for the operational semantics of stateful languages, and equipped it with three semantics: resumption semantics, trace semantics, and termination semantics, in decreasing order of granularity. Our main interest has been in compositionality of these semantics. While resumption semantics is always compositional, it is in general undecidable whether the coarser semantics are compositional. However, compositionality is ensured by restricting to *streamlined* stateful SOS specifications for trace semantics, and to *cool* stateful SOS specifications for termination semantics. The compositionality result for the cool format improves on previous results for the similar *evaluation-in-context* formats [2] by abstracting from steps until termination. The streamlined format is more permissive, as we illustrate on a signal handling construct.

Our results currently work with deterministic state transformers, captured by the functor $TX = (BX)^S$ where $BX = S \times (X+1)$. We believe that our results generalize to functors $B$ equipped with a natural transformation $c_X\colon BX \to S$. As a first step, this generalization requires an abstract characterization of our streamlined and cool rule formats in terms of their corresponding natural transformations, along with categorical proofs of the respective congruence theorems. We leave this as an important point for future work.

A further direction of possible generalization is to cover *effects*, such as non-determinism, in a similar style as in work on evaluation-in-context [2]. Our work embeds the standard semantics of sequential imperative programming (in particular termination semantics) into the paradigm of operational semantics via distributive laws, and we expect to relate our

results to work on morphisms of distributive laws [40, 20], which, for instance, have recently been shown to have applications to secure compilation [35]. Extending the overall paradigm to support higher-order languages is a well-known and, so far, elusive problem. Like in the current work, tackling this problem may require a slight deviation from the standard form of GSOS laws. It is worth noting that rule formats for higher-order languages have been proposed in the past by Howe [15], Bernstein [6] and more recently Hirschowitz and Lafont [14].

Our treatment of resumption and trace semantics and their relationship is generic, and presumably can be transferred to other settings, in particular to constructive and type-theoretic frameworks. Indeed we expect that it can be implemented relatively directly in foundational proof assistants such as Agda, without additional postulates (such as the axiom of choice or the law of excluded middle). In contrast, the domain $(S+1)^S$ of termination semantics is inherently classical, as it postulates that every computation will either terminate or diverge. This can be remedied by replacing the maybe-monad $(-)+1$ with a suitable *partiality monad* [3, 11]. We will explore to what extent our results regarding termination semantics can be rebased on this more general perspective.

## References


1. Faris Abou-Saleh and Dirk Pattinson. Towards effects in mathematical operational semantics. In Michael W. Mislove and Joël Ouaknine, editors, *Mathematical Foundations of Programming Semantics, MFPS 2011*, volume 276 of *Electron. Notes Theor. Comput. Sci.*, pages 81–104. Elsevier, 2011. doi:10.1016/j.entcs.2011.09.016.
2. Faris Abou-Saleh and Dirk Pattinson. Comodels and effects in mathematical operational semantics. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7794 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2013. doi:10.1007/978-3-642-37075-5_9.
3. Thorsten Altenkirch, Nils Danielsson, and Nicolai Kraus. Partiality, revisited - the partiality monad as a quotient inductive-inductive type. In Javier Esparza and Andrzej Murawski, editors, *Foundations of Software Science and Computation Structures, FOSSACS 2017*, volume 10203 of *Lecture Notes Comput. Sci.*, pages 534–549, 2017.
4. Michael Barr. Coequalizers and free triples. *Math. Z.*, 116:307–322, 1970.
5. Falk Bartels. *On generalised coinduction and probabilistic specification formats: Distributive laws in coalgebraic modelling.* PhD thesis, Vrije Universiteit Amsterdam, 2004.
6. Karen L. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 153–164. IEEE Computer Society, 1998. doi:10.1109/LICS.1998.705652.
7. Bard Bloom. Structural operational semantics for weak bisimulations. *Theor. Comput. Sci.*, 146(1&2):25–68, 1995. doi:10.1016/0304-3975(94)00152-9.
8. Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995. doi:10.1145/200836.200876.
9. Bard Bloom and Frits Vandraager. Sos rule formats for parameterized and state-bearing processes, 1994. URL: http://www.sws.cs.ru.nl/publications/papers/fvaan/bardfrits.ps.
10. Stephen D. Brookes. Full abstraction for a shared variable parallel language. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 98–109. IEEE Computer Society, 1993. doi:10.1109/LICS.1993.287596.

11 James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science*, 29(1):67–92, 2019.

12 Marcelo P. Fiore and Sam Staton. A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 49–58. IEEE Computer Society, 2006. doi:10.1109/LICS.2006.7.

13 Matthew Hennessy and Gordon D. Plotkin. Full abstraction for a simple parallel programming language. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 1979. doi:10.1007/3-540-09526-8_8.

14 Tom Hirschowitz and Ambroise Lafont. A categorical framework for congruence of applicative bisimilarity in higher-order languages. *CoRR*, abs/2103.16833, 2021. arXiv:2103.16833.

15 Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2):103–112, 1996. doi:10.1006/inco.1996.0008.

16 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.

17 Alan Jeffrey and Julian Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In Shmuel Sagiv, editor, *14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2005. doi:10.1007/978-3-540-31987-0_29.

18 Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. doi:10.1007/BFb0039592.

19 Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011. doi:10.1016/j.tcs.2011.03.023.

20 Bartek Klin and Beata Nachyla. Presenting morphisms of distributive laws. In *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015, June 24-26, 2015, Nijmegen, The Netherlands*, pages 190–204, 2015. doi:10.4230/LIPIcs.CALCO.2015.190.

21 Bartek Klin and Vladimiro Sassone. Structural operational semantics for stochastic process calculi. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2008. doi:10.1007/978-3-540-78499-9_30.

22 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008. arXiv:0808.0586.

23 Marino Miculan and Marco Peressotti. Structural operational semantics for non-deterministic processes with quantitative aspects. *Theor. Comput. Sci.*, 655:135–154, 2016. doi:10.1016/j.tcs.2016.01.012.

24 Mohammad Reza Mousavi, Michel Reniers, and Jan Friso Groote. Congruence for sos with data. In *LICS*, pages 302–313. IEEE Computer Society Press, 2004.

25 Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2009. doi:10.1007/978-3-642-03359-9_26.

26 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016,*

*Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

**27** Marco Patrignani and Dave Clarke. Fully abstract trace semantics for protected module architectures. *Comput. Lang. Syst. Struct.*, 42:22–45, 2015. doi:10.1016/j.cl.2015.03.002.

**28** Marco Patrignani, Dominique Devriese, and Frank Piessens. On modular and fully-abstract compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 17–30. IEEE Computer Society, 2016. doi:10.1109/CSF.2016.9.

**29** Maciej Piróg and Jeremy Gibbons. Monads for behaviour. In *Mathematical Foundations of Programming Semantics, MFPS 2013*, volume 298 of *Electron. Notes Theor. Comput. Sci.*, pages 309–324, 2015.

**30** Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000. doi:10.1007/3-540-45699-6_8.

**31** Andrew M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–274. Cambridge University Press, New York, NY, USA, 1998.

**32** Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

**33** Jan J. M. M. Rutten. A note on coinduction and weak bisimilarity for while programs. *ITA*, 33(4/5):393–400, 1999. doi:10.1051/ita:1999125.

**34** Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

**35** Stelios Tsampas, Andreas Nuyts, Dominique Devriese, and Frank Piessens. A categorical approach to secure compilation. In Daniela Petrisan and Jurriaan Rot, editors, *Coalgebraic Methods in Computer Science - 15th IFIP WG 1.3 International Workshop, CMCS 2020, Colocated with ETAPS 2020, Dublin, Ireland, April 25-26, 2020, Proceedings*, volume 12094 of *Lecture Notes in Computer Science*, pages 155–179. Springer, 2020. doi:10.1007/978-3-030-57201-3_9.

**36** Stelios Tsampas, Christian Williams, Andreas Nuyts, Dominique Devriese, and Frank Piessens. Abstract congruence criteria for weak bisimilarity. In Filippo Bonchi and Simon J. Puglisi, editors, *46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23-27, 2021, Tallinn, Estonia*, volume 202 of *LIPIcs*, pages 88:1–88:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.MFCS.2021.88.

**37** Daniele Turi. Categorical modelling of structural operational rules: Case studies. In *Category Theory and Computer Science, 7th International Conference, CTCS '97, Santa Margherita Ligure, Italy, September 4-6, 1997, Proceedings*, pages 127–146, 1997. doi:10.1007/BFb0026985.

**38** Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 280–291, 1997. doi:10.1109/LICS.1997.614955.

**39** Rob J. van Glabbeek. On cool congruence formats for weak bisimulations. *Theor. Comput. Sci.*, 412(28):3283–3302, 2011. doi:10.1016/j.tcs.2011.02.036.

**40** Hiroshi Watanabe. Well-behaved translations between structural operational semantics. *Electr. Notes Theor. Comput. Sci.*, 65(1):337–357, 2002. doi:10.1016/S1571-0661(04)80372-4.

# A Combinatorial Approach to Higher-Order Structure for Polynomial Functors

**Marcelo Fiore**
University of Cambridge, UK

**Zeinab Galal**
University of Leeds, UK

**Hugo Paquet**
University of Oxford, UK

## Abstract

Polynomial functors are categorical structures used in a variety of applications across theoretical computer science; for instance, in database theory, denotational semantics, functional programming, and type theory. A well-known problem is that the bicategory of finitary polynomial functors between categories of indexed sets is not cartesian closed, despite its success and influence on denotational models and linear logic.

This paper introduces a formal bridge between the model of finitary polynomial functors and the combinatorial theory of generalised species of structures. Our approach consists in viewing finitary polynomial functors as free analytic functors, which correspond to free generalised species. In order to systematically consider finitary polynomial functors from this combinatorial perspective, we study a model of groupoids with additional logical structure; this is used to constrain the generalised species between them. The result is a new cartesian closed bicategory that embeds finitary polynomial functors.

**2012 ACM Subject Classification** Theory of computation → Categorical semantics; Theory of computation → Lambda calculus; Mathematics of computing → Combinatorics

**Keywords and phrases** Bicategorical models, denotational semantics, stable domain theory, linear logic, polynomial functors, species of structures, groupoids

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.31

**Funding** *Marcelo Fiore*: Research partially supported by EPSRC grant EP/V002309/1.
*Zeinab Galal*: Research partially supported by EPSRC grant EP/V002309/1 and by ANR grant ANR-20-CE48-0010.
*Hugo Paquet*: Research supported by a Royal Society University Research Fellowship.

## 1 Introduction

We introduce a formal bridge between two mathematical theories which have been influential in the context of programming language semantics:

1. The theory of *polynomial functors*, a popular categorification of the notion of polynomial function.
2. The theory of *generalised species* and analytic functors, due to Fiore, Gambino, Hyland, and Winskel, which provides higher-order notions of combinatorial structures.

The connection gives a new combinatorial perspective on polynomial functors. We exploit this in the paper to overcome the problem that polynomial functors do not form a cartesian closed model.

© Marcelo Fiore, Zeinab Galal, and Hugo Paquet;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 31; pp. 31:1–31:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Applications of polynomial functors in computer science are surprisingly varied, and include: models of dependent type theory [21, 48, 5], representation of data types [1, 4], implicit complexity theory [41], and dynamical systems [43]. As for semantics, polynomial functors support a well-known model of the lambda calculus, developed by Girard [26], who explicitly cites it as a catalyst for linear logic [25].

Generalised species [18] were put forward more recently as a general *bicategorical* framework for the study of substitution for combinatorial structures, generalising Joyal's prior work on species of structures [31, 32]. The bicategory of generalised species is cartesian closed, and thus provides a convenient basis for denotational semantics. It has become a prime example of a semantic model in which program symmetries are represented explicitly as 2-cells, and several lines of research are benefitting from this idea [47, 19, 40, 20, 42].

The connection between these two concepts is already understood in simple settings; we give a brief overview. On one side, we consider *finitary* polynomial functors $\mathbf{Set} \longrightarrow \mathbf{Set}$, which correspond to operations on sets of the form

$$X \longmapsto \sum_{n\in\mathbb{N}} A_n \times X^n \tag{1}$$

where the coefficients $A_n$ are sets. On the other side, Joyal's species of structures are equivalent to *analytic functors* $\mathbf{Set} \longrightarrow \mathbf{Set}$, corresponding to operations of the form

$$X \longmapsto \sum_{n\in\mathbb{N}} F_n \underset{\mathfrak{S}_n}{\times} X^n \tag{2}$$

where the coefficients $F_n$ are sets equipped with an action of the symmetric group $\mathfrak{S}_n$ on $n$ elements, and the operator $\underset{\mathfrak{S}_n}{\times}$ performs a quotient of the product under this action. In special cases, when the actions on $F_n$ are *free actions* (§5), the quotient is equivalently a set of the form $A_n \times X^n$, and so the analytic functor is also polynomial. Conversely, every finitary polynomial functor is analytic when its coefficients are regarded as freely generated actions.

## Summary of contributions

We extend the correspondence between finitary polynomial functors and free analytic functors to a generalised setting: instead of functors between categories of indexed sets, we consider functors between full subcategories of presheaves over groupoids. Our first contribution is a logical device for constraining the actions on the coefficients of analytic functors: in particular one may require all actions to be free. We call this device a *kit* (§3).

We show that one can systematically consider analytic functors controlled by kits. This leads us to the construction of a 2-category whose morphisms we called *stable functors*. In the basic setting of endofunctors on sets, we recover the simple connection above: stable functors correspond to finitary polynomial functors.

We then push this further and consider higher-order structure in this bicategory. We introduce *stable species*, combinatorial structures constrained by kits, and show that these correspond to stable functors, just as generalised species correspond to analytic functors. This gives our second main contribution: we prove that stable species, and therefore the equivalent stable functors, form a cartesian closed bicategory.

This is significant because the bicategory of finitary polynomial functors between categories of indexed sets is not cartesian closed; for instance, Girard's lambda-calculus model cannot be extended directly to a typed lambda calculus. This situation has attracted a considerable amount of attention [45, 28, 12]; our approach has the advantage of making the combinatorics of the problem clear and explicit, via the kit on the function space in our bicategory.

### Outline of the paper

We first (§2) give an introductory account of the connection between analytic and polynomial endofunctors on sets, including the representation of coefficients as species. Then, the formal development is organised as follows:

- We introduce *groupoids with kits*, demonstrating their purpose in controlling actions (§3) and identifying the important class of *Boolean kits*.
- We introduce a 2-category **Stable** whose objects are groupoids with Boolean kits and whose morphisms are called *stable functors* (§4.3). Stable functors are closely related to finitary polynomial functors, coinciding with them at discrete groupoids.
- We introduce a bicategory **SEsp** of *stable species of structures* (§6), a refinement of generalised species of structures, based on groupoids with Boolean kits. We establish that **SEsp** is cartesian closed (Theorem 22).
- We exhibit a biequivalence between **SEsp** and **Stable** (§7), deducing that **Stable** is a cartesian closed bicategory (Theorem 17).

Finally (§8) we mention related work in the area. We explain the influence of Taylor's *creeds* [45] on our work, and discuss connections with Berry's stable domain theory [9] and with Girard's linear logic [25].

## 2 Polynomial functors and analytic endofunctors on sets

At the simplest level, polynomial and analytic functors are defined on the category **Set** of sets and functions.

### Polynomial functors

A function $p : E \to B$ between sets $E$ and $B$ determines an endofunctor on **Set** defined as

$$X \longmapsto \sum_{b \in B} X^{E_b}$$

where $E_b = p^{-1}\{b\}$ is the fibre of $p$ over $b \in B$, and $X^{E_b}$ is the set of functions $E_b \to X$. It is common (although not essential for this paper) to think of $B$ as a set of operators, where the arity of an operator $b \in B$ is specified by (the cardinality of) $E_b$. In particular, one can restrict to *finitary* polynomial functors with finite arities. Every finitary polynomial functor is then naturally isomorphic to one of the form

$$X \longmapsto \sum_{n \in \mathbb{N}} F_n \times X^n \tag{3}$$

where each set $F_n$ corresponds to the set of operators of arity $n$. The analogy with traditional polynomials is manifest in this representation. Finitary polynomial functors are determined by their action on finite input sets, in the same vein as continuous maps between domains in domain theory.

### Analytic functors and Joyal species of structures

An endofunctor on **Set** is an *analytic functor* [32] if it is naturally isomorphic to one of the form

$$X \longmapsto \sum_{n \in \mathbb{N}} F(n) \underset{\mathfrak{S}_n}{\times} X^n, \tag{4}$$

where:

1. Each $F(n)$ is a set with a left action of the symmetric group $\mathfrak{S}_n$ on $n$ elements. Concretely, this means that we have an assignment that to every permutation $\sigma \in \mathfrak{S}_n$ of the set $[n] = \{1, \dots, n\}$ associates a permutation of the set $F(n)$ preserving identity and composition. We write $\sigma \cdot p$ for the action of $\sigma \in \mathfrak{S}_n$ on an element $p \in F(n)$.
2. The set $F(n) \underset{\mathfrak{S}_n}{\times} X^n$ is obtained by quotienting the product $F(n) \times X^n$ under the equivalence relation $\sim$ containing the pairs

$$(p, (x_{\sigma 1}, \dots, x_{\sigma n})) \;\sim\; (\sigma \cdot p, (x_1, \dots, x_n))$$

for all $\sigma \in \mathfrak{S}_n$, $p \in F(n)$ and $(x_1, \dots, x_n) \in X^n$.

The notation $F(-)$ is justified, because the coefficients $F(n)$, *together with* the group actions, can be bundled into a functor $F : \mathbf{B} \longrightarrow \mathbf{Set}$ where $\mathbf{B}$ is the category whose objects are the natural numbers, and whose morphisms $m \to n$ are the bijections $[m] \to [n]$. The action of a permutation $\sigma \in \mathfrak{S}_n$ on the set $F(n)$ is then simply given by the functorial action $F(\sigma) : F(n) \to F(n)$.

The functor $F : \mathbf{B} \longrightarrow \mathbf{Set}$ is a *species of structures* (or just a *species*, with its elements referred to as *structures*) corresponding to the analytic functor (4). Every analytic functor has, up to isomorphism, a unique generating species, which may be recovered using so-called *weak generic elements* (§7). This combinatorial theory was developed by Joyal [31, 32], including the connection to polynomial functors as we explain next.

## Polynomial functors are free analytic functors

Finitary polynomial functors correspond to free analytic functors and this gives an equivalence. The basic idea is as follows. Every set $A$ generates a *free action* of a group $G$, given by the product set $A \times G$ with the action

$$\tau \cdot (a, \sigma) \stackrel{\text{def}}{=} (a, \tau\,\sigma) \tag{5}$$

for every $\tau \in G$ and $(a, \sigma) \in A \times G$. We extend this to polynomial functors. Consider the polynomial functor $X \longmapsto \sum_{n \in \mathbb{N}} A_n \times X^n$. Taking the free action generated by $A_n$ of $\mathfrak{S}_n$, for every $n \in \mathbb{N}$, we obtain a species $\mathbf{B} \longrightarrow \mathbf{Set}$ given by

$$\begin{aligned} n &\longmapsto A_n \times \mathfrak{S}_n \\ (\tau : n \to n) &\longmapsto ((a, \sigma) \mapsto (a, \tau\,\sigma)) \end{aligned}$$

which, via the construction (4), generates an analytic functor. For this species, when taking the quotient in (4), we have

$$(A_n \times \mathfrak{S}_n) \underset{\mathfrak{S}_n}{\times} X^n \;\cong\; A_n \times X^n$$

and therefore recover the polynomial functor. Thus, every finitary polynomial functor is, in particular, analytic.

One can characterize the analytic functors that are polynomial in terms of the generating species. To do this, observe the following key property: for the free action defined in (5), every element has trivial stabilizer. Recall that the *stabilizer* of an element $p \in P$ with respect to the action of a group $G$ is defined as $\mathrm{Stab}_G(p) \stackrel{\text{def}}{=} \{\sigma \in G \mid \sigma \cdot p = p\}$, a subgroup of $G$. Then, if $F : \mathbf{B} \longrightarrow \mathbf{Set}$ is a species such that for every $n \in \mathbb{N}$, the action of $\mathfrak{S}_n$ on $F(n)$ is free (in the sense that every structure in $F(n)$ has trivial stabilizer) then the associated analytic endofunctor on $\mathbf{Set}$ is polynomial.

### Cartesian natural transformations between polynomial functors

We have described finitary polynomial functors as a subclass of analytic functors. Following Girard and others [26, 45], the natural transformations to be considered between them are as follows:

▶ **Definition 1.** *A* ***cartesian natural transformation*** *is a natural transformation for which every naturality square is a pullback.*

There are several justifications for this choice:

- With the interpretation of polynomial functors as arising from sets of operators with arities, a cartesian natural transformation corresponds to a mapping between operator sets that preserves arities.
- Cartesian natural transformations between polynomial functors are in bijection with (arbitrary) natural transformations between the associated free species.
- Cartesian natural transformations are a categorification of Berry's stable order in domain theory, a point of view that motivates our approach (see the discussion in §8).

### The combinatorial and extensional views

So far, we have given definitions of polynomial and analytic functors in terms of coefficient species. This is the *combinatorial* (or *intensional*) view. A strength of the theory is that there is an alternative presentation: both kinds of functors may be characterised abstractly without reference to coefficients. This is the *extensional* view:

- Analytic endofunctors on sets are the finitary ones (i.e. filtered-colimit preserving) that preserve wide quasi-pullbacks [32].
- Polynomial endofunctors on sets are those that preserve wide pullbacks (equivalently, enjoy a local right adjoint property, see Definition 5) [44, 22].

We will generalise beyond endofunctors on sets and define our *stable functors* in the extensional style. But it is the combinatorial view, given by *stable species*, that explains the higher-order structure.

## 3 Groupoids with kits

We introduce kits, a structure for controlling group (and, more generally, groupoid) actions. As motivation for the general theory, the discussion in this section is only concerned with endofunctors on sets.

### Species and stabilizers

Recall two key observations from the previous section:

- every species $\mathbf{B} \longrightarrow \mathbf{Set}$ induces an analytic endofunctor on sets; and
- this functor is polynomial if and only if the species is *free* (in the sense that all its structures have trivial stabilizer).

One key idea of this paper is to use subgroups to specify the extent to which a species may be free. Indeed, by specifying, for each $n \in \mathbf{B}$, a set $\mathcal{K}(n)$ of subgroups of $\mathbf{B}(n, n)$ that are to be regarded as *permitted stabilizers*, we may restrict to species with structures having only permitted stabilizers (Definition 4) and thereby identify a class of generalised polynomial functors. Appropriate such families $\mathcal{K} = \{\mathcal{K}(n)\}_{n \in \mathbf{B}}$ we will call kits (Definition 2).

As extreme special cases, one can take $\mathcal{K}(n)$ to contain all the subgroups of $\mathbf{B}(n,n)$ and recover the analytic functors; or, instead, take $\mathcal{K}(n)$ to consist only of the trivial subgroup, forcing the species to be free, and recover polynomial functors.

There is no need that the choice of permitted stabilizers be uniform across all $n \in \mathbf{B}$ as in the two examples above. But it is natural to require that permitted stabilizers are closed under conjugation, since for every structure $p \in F(n)$ and permutation $\sigma \in \mathbf{B}(n,n)$, the stabilizer subgroups of $p$ and of $\sigma \cdot p$ are conjugate of each other: $\mathrm{Stab}(\sigma \cdot p) = \sigma\ \mathrm{Stab}(p)\ \sigma^{-1}$.

As a step towards the generalised model to come, we introduce kits on arbitrary groupoids. This brings us close to Taylor's *creeds* [45], see also §8.

▶ **Definition 2.** *A* ***kit*** *on a groupoid $\mathbb{A}$ is a family $\mathcal{A} = \{\, \mathcal{A}(a)\, \}_{a\in\mathbb{A}}$ where $\mathcal{A}(a)$ is a set of subgroups of $\mathbb{A}(a,a)$ closed under conjugation in the following sense:*

*For all $a, a' \in \mathbb{A}$ and $\alpha \in \mathbb{A}(a,a')$, if $H \in \mathcal{A}(a)$ then $\alpha\, H\, \alpha^{-1} \in \mathcal{A}(a')$.*

The proposition below provides important examples of kits.

▶ **Proposition 3.** *For a presheaf $F : \mathbb{A}^{\mathrm{op}} \longrightarrow \mathbf{Set}$ on a groupoid $\mathbb{A}$, the family $\mathcal{S} = \{\, \mathcal{S}(a)\, \}_{a\in\mathbb{A}}$ with $\mathcal{S}(a) = \{\, \mathrm{Stab}_{\mathbb{A}(a,a)}(p) \mid p \in F(a)\, \}$ is a kit.*

At this stage, we are in a position to define a restricted notion of analytic endofunctor on **Set** parametrised by a kit on the groupoid $\mathbf{B}$. First we appropriately restrict species:

▶ **Definition 4.** *Let $\mathcal{K}$ be a kit on $\mathbf{B}$. A* ***$\mathcal{K}$-species*** *is a functor $F : \mathbf{B} \longrightarrow \mathbf{Set}$ such that every $F$-structure has stabilizer in $\mathcal{K}$.*

This way, every kit $\mathcal{K}$ gives rise to the subclass of analytic endofunctors on **Set** induced by $\mathcal{K}$-species. In this paper, we are targeting polynomial functors and, in accordance, our construction of stable species (Definition 19) induces the kit on $\mathbf{B}$ that consists of only the trivial subgroups.

## 4 Stable functors between categories of stable presheaves

A kit $\mathcal{A}$ on a groupoid $\mathbb{A}$ determines a full subcategory $\boldsymbol{\mathcal{S}}(\mathbb{A},\mathcal{A})$ of the presheaf category $\boldsymbol{\mathcal{P}}(\mathbb{A}) = [\mathbb{A}^{\mathrm{op}}, \mathbf{Set}]$, whose objects we call *stable presheaves*. We will consider *stable functors* between categories of stable presheaves. These generalise finitary polynomial functors beyond endofunctors on **Set**. We begin with a brief overview of existing generalisations of polynomial functors and analytic functors which are relevant to the paper.

### 4.1 Generalising polynomial and analytic functors

#### Polynomial functors between categories of indexed sets

There is a rich theory of polynomial functors defined with respect to a *locally cartesian closed* category [22, 4]. When that category is **Set**, one obtains a notion of polynomial functor $\mathbf{Set}^I \longrightarrow \mathbf{Set}^J$, where $I$ and $J$ are sets. These can be given representations with coefficients in the style of (3). As we will later on study these coefficients using generalised species, for now we give a definition in the extensional style [22, §1.18]. This will form the basis for our notion of stable functor (Definition 10).

▶ **Definition 5.** *A functor $F : \mathscr{C} \longrightarrow \mathscr{D}$ is a* ***local right adjoint*** *if for every object $C \in \mathscr{C}$, the local functor*

$$F/C : \mathscr{C}/C \longrightarrow \mathscr{D}/F(C)$$

*between slice categories, which transports an object $A \xrightarrow{a} C$ to its image $F(A) \xrightarrow{F(a)} F(C)$, has a left adjoint.*

*For sets $I$ and $J$, a functor $\mathbf{Set}^I \longrightarrow \mathbf{Set}^J$ is a* ***polynomial functor*** *if it is a local right adjoint.*

Polynomial functors between categories of indexed sets can be organised into a 2-category with indexing sets as objects, and cartesian natural transformations as 2-cells. This model has convenient structure: for instance, there is a canonical polynomial functor isomorphism $\mathbf{Set}^{I+J} \cong \mathbf{Set}^I \times \mathbf{Set}^J$ that exhibits $I + J$ as a cartesian product of $I$ and $J$. Restricting to finitary polynomial functors, Girard (who called them *normal functors*) described a model of the lambda calculus [26]. His idea of representing programs as polynomials triggered a radical shift in perspective, leading to linear logic [25].

However, the bicategory of polynomial functors is not cartesian closed. One approach to circumvent this, pioneered by Lamarche [36], is to consider polynomial *functions* between domains rather than categories, where coefficients are elements of a suitable semiring. Here we propose a solution using generalised species of structures, which we present next.

### Generalised species of structures and analytic functors

We consider analytic functors between presheaf categories over groupoids. These include categories of indexed sets $\mathbf{Set}^I$, viewing the set $I$ as a discrete groupoid.

Recall the notion of a species $\mathbf{B} \longrightarrow \mathbf{Set}$ for an analytic functor on $\mathbf{Set}$ and consider the following two basic observations. First, the groupoid $\mathbf{B}$ is the symmetric strict monoidal completion $!\mathbf{1}$ of the terminal category $\mathbf{1}$ with one object and one morphism. Second, $\mathbf{Set}$ is isomorphic to the category of presheaves $\boldsymbol{\mathcal{P}}\mathbf{1}$ over $\mathbf{1}$. The approach of Fiore, Gambino, Hyland and Winskel [18] extends the basic notion of a species $\mathbf{B} \longrightarrow \mathbf{Set}$, corresponding to $!\mathbf{1} \longrightarrow \boldsymbol{\mathcal{P}}\mathbf{1}$, to a *generalised species*

$$!\mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}\mathbb{B} \tag{6}$$

for $\mathbb{A}$ and $\mathbb{B}$ groupoids (in fact, they can be arbitrary small categories but we do not use this generality here). Their main result is that these assemble into a bicategory of groupoids, generalised species, and natural transformations that is cartesian closed.

A generalised species $!\mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}\mathbb{B}$ induces an analytic functor $\boldsymbol{\mathcal{P}}\mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}\mathbb{B}$ between presheaf categories which we will recall in §7. This generalises Joyal's notion (4) to analytic functors between presheaf categories. These analytic functors have also been characterised extensionally [16].

## 4.2 Stable presheaves

We move towards the construction of a 2-category (Proposition 16) whose objects are groupoids with appropriate kits and whose morphisms are functors between full subcategories of presheaves with permitted stabilizers in kits (or *quantitative domains* [45]).

▶ **Definition 6.** *Let $\mathbb{A}$ be a groupoid equipped with a kit $\mathcal{A}$. A presheaf $F : \mathbb{A}^{\mathrm{op}} \longrightarrow \mathbf{Set}$ is an* ***$\mathcal{A}$-stable presheaf*** *if every element of $F$ has stabilizer in $\mathcal{A}$. The* ***category of $\mathcal{A}$-stable presheaves on*** *$\mathbb{A}$, a full subcategory of $\boldsymbol{\mathcal{P}}(\mathbb{A})$, is denoted $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$.*

Observe, in particular, that one may recover the whole presheaf category by permitting all subgroups: $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) = \boldsymbol{\mathcal{P}}(\mathbb{A})$ for the maximal kit $\mathcal{A}$ (that is, the one consisting of all subgroups of endomorphisms).

The category $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$ has a convenient and intuitive characterisation in terms of sums and quotients, which fits well with our goal of controlling quotients in polynomials. In the discussion below, we will use the following basic elements from the theory of presheaves:

- The *representable* presheaves on $\mathbb{A}$ are those which are naturally isomorphic to ones of the form $\mathbb{A}(-, a) : \mathbb{A}^{\mathrm{op}} \longrightarrow \mathbf{Set}$ for some object $a \in \mathbb{A}$.
- There is a functor $\mathrm{y} : \mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}(\mathbb{A}) : a \mapsto \mathbb{A}(-, a)$, the *Yoneda embedding*, that is full and faithful.
- Presheaf categories have all small limits and colimits, which are calculated pointwise in terms of those in **Set**.
- Every presheaf is a canonical colimit of representable presheaves. In fact, the Yoneda embedding exhibits $\boldsymbol{\mathcal{P}}\mathbb{A}$ as the small colimit completion of $\mathbb{A}$.
- For presheaves on groupoids this last point can be strengthened considerably: every presheaf on a groupoid is a sum of quotients of representable presheaves by subgroups. In fact, for a groupoid $\mathbb{A}$, $\boldsymbol{\mathcal{P}}\mathbb{A}$ is the coproduct completion of the quotients of representable presheaves by subgroups (Theorem 9).

The above quotients are special colimits as follows: for an object $a$ of a groupoid $\mathbb{A}$, the quotient of $\mathrm{y}(a)$ by a subgroup $H$ of $\mathbb{A}(a, a)$ is the colimit $q : \mathrm{y}(a) \to \mathrm{y}(a)_{/H}$ of the diagram $H \hookrightarrow \mathbb{A} \xrightarrow{\mathrm{y}} \boldsymbol{\mathcal{P}}\mathbb{A}$ as depicted below:

$$\mathrm{y}(a) \overset{q}{\dashrightarrow} \mathrm{y}(a)_{/H} \qquad \text{(loop on } \mathrm{y}(a) \text{ labelled } \underset{(h \in H)}{\mathrm{y}(h)}\text{)} \tag{7}$$

Concretely, the presheaf $\mathrm{y}(a)_{/H}$ maps an object $x$ to the quotient of $\mathrm{y}(a)(x) = \mathbb{A}(x, a)$ under the equivalence relation $\sim_H$ given by $\alpha' \sim_H \alpha$ if and only if $\alpha' \alpha^{-1} \in H$. Quotienting under the trivial subgroup has no effect: $\mathrm{y}(a)_{/\{\mathrm{id}_a\}} \cong \mathrm{y}(a)$; while quotienting under the full group of endomorphisms gives the presheaf $\mathrm{y}(a)_{/\mathbb{A}(a,a)}$ that maps $x$ to a singleton when $x \cong a$ and to the empty set otherwise.

Quotients and stabilizers are closely related.

▶ **Proposition 7.** *Let $\mathbb{A}$ be a groupoid and let $H$ be a subgroup of $\mathbb{A}(a, a)$ for $a \in \mathbb{A}$. For all $x \in \mathbb{A}$ and $\alpha \in \mathrm{y}(a)_{/H}(x)$, $\mathrm{Stab}(\alpha) = H$.*

In particular, quotients of representable presheaves by subgroups in a kit are stable presheaves:

▶ **Corollary 8.** *Let $\mathcal{A}$ be a kit on a groupoid $\mathbb{A}$. For $a \in \mathbb{A}$ and $H \in \mathcal{A}(a)$, $\mathrm{y}(a)_{/H} \in \boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$.*

More generally, we have a representation theorem for stable presheaves as follows:

▶ **Theorem 9.** *Let $\mathbb{A}$ be a groupoid equipped with a kit $\mathcal{A}$. Then, assuming the axiom of choice, every presheaf $X$ in $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$ is isomorphic to a sum of quotients of representable presheaves by subgroups in $\mathcal{A}$; that is,*

$$X \cong \sum_{i \in I} \mathrm{y}(a_i)_{/H_i}$$

*for some $I$-indexed family $\{ (a_i, H_i) \}_{i \in I}$ of objects $a_i \in \mathbb{A}$ and groups $H_i \in \mathcal{A}(a_i)$. Conversely, every presheaf of this form is in $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$.*

Therefore, kits provide a concrete way of restricting the coproduct completion of quotients of representable presheaves by subgroups, yielding the stable presheaves.

## 4.3 Stable functors

We will consider *stable functors* (Definition 10) of type $\boldsymbol{\mathcal{S}}(\mathbb{A},\mathcal{A}) \longrightarrow \boldsymbol{\mathcal{S}}(\mathbb{B},\mathcal{B})$, our notion of finitary polynomial functor generalised to categories of stable presheaves. Our definition is extensional and abstract enough to be stated generally:

▶ **Definition 10.** *Let $\mathscr{C}$ and $\mathscr{D}$ be categories. We call a functor $F : \mathscr{C} \longrightarrow \mathscr{D}$* ***stable*** *if it satisfies the following conditions:*

- *$F$ is a local right adjoint.*
- *$F$ is finitary (that is, preserves filtered colimits).*
- *$F$ preserves regular epimorphisms.*

We comment on the definition. The local right adjoint condition is an extensional presentation of polynomial functors (recall Definition 5). The restriction to finitary functors is natural in the context of semantic models of higher-order computation; it is also essential because our proof of cartesian closure (Theorems 22 and 32) of the bicategory **Stable** (introduced in Proposition 16 below) is based on a representation by means of finitary coefficients in the style of (4) and (6). The preservation of regular epimorphisms crucially ensures the preservation of quotient maps (7). This condition is not necessary for finitary polynomial functors of type $\mathbf{Set}^I \longrightarrow \mathbf{Set}^J$, which always preserve regular epimorphisms (as do arbitrary such polynomial functors assuming the axiom of choice).

▶ **Example 11.** In a cartesian closed and extensive category, such as a topos, the finite product and finite coproduct functors are stable. In connection to this, we discuss the prototypical non-stable, and hence non-sequential, function from Berry's stable domain theory [9]. To this end, let $S$ be the Sierpinski space $(0 \subset 1)$ and, for $\mathrm{Bool} = \{\mathbf{f}, \mathbf{t}\}$, consider the *parallel-or* function $por : S^{\mathrm{Bool}} \times S^{\mathrm{Bool}} \to S^{\mathrm{Bool}}$ defined as the least monotone function such that: $por(\{\mathbf{f}\},\{\mathbf{f}\}) = \{\mathbf{f}\}$ and $por(\{\mathbf{t}\},\{\,\}) = por(\{\,\},\{\mathbf{t}\}) = \{\mathbf{t}\}$.

Parallel-or is not realisable as a stable functor at the categorical level in the strong sense that there is no stable functor $F : \mathbf{Set}^{\mathrm{Bool}} \times \mathbf{Set}^{\mathrm{Bool}} \longrightarrow \mathbf{Set}^{\mathrm{Bool}}$ such that $F(0,0) = 0$ and $F(\mathrm{y}(\mathbf{t}),0) = F(0,\mathrm{y}(\mathbf{t})) = F(\mathrm{y}(\mathbf{t}),\mathrm{y}(\mathbf{t})) = \mathrm{y}(\mathbf{t})$. Indeed, such functors do not preserve the pullback

$$\begin{array}{ccccc} & & (\mathrm{y}(\mathbf{t}),\mathrm{y}(\mathbf{t})) & & \\ & \nearrow & & \nwarrow & \\ (\mathrm{y}(\mathbf{t}),0) & & & & (0,\mathrm{y}(\mathbf{t})) \\ & \nwarrow & & \nearrow & \\ & & (0,0) & & \end{array}$$

and thus induce local functors $F/(\mathrm{y}(\mathbf{t}),\mathrm{y}(\mathbf{t})) : \mathbf{Set}^{\mathrm{Bool}} \times \mathbf{Set}^{\mathrm{Bool}}/(\mathrm{y}(\mathbf{t}),\mathrm{y}(\mathbf{t})) \longrightarrow \mathbf{Set}^{\mathrm{Bool}}/\mathrm{y}(\mathbf{t})$ that fail to be right adjoints.

However, the generalisation from domains to categories allows for an intensional quantitative interpretation of parallel or. Indeed, for $K : \mathbf{Set} \longrightarrow S$ the *collapse* functor mapping a set to 0 if it is empty and to 1 otherwise, we have a stable functor

$$P(X,Y) = (X_{\mathbf{f}} \times Y_{\mathbf{f}}, X_{\mathbf{t}} + Y_{\mathbf{t}})$$

lifting *por* as follows:

$$\begin{array}{ccc} \mathbf{Set}^{\mathrm{Bool}} \times \mathbf{Set}^{\mathrm{Bool}} & \xrightarrow{P} & \mathbf{Set}^{\mathrm{Bool}} \\ {\scriptstyle K^{\mathrm{Bool}} \times K^{\mathrm{Bool}}} \downarrow & & \downarrow {\scriptstyle K^{\mathrm{Bool}}} \\ S^{\mathrm{Bool}} \times S^{\mathrm{Bool}} & \xrightarrow[por]{} & S^{\mathrm{Bool}} \end{array}$$

# 5 Boolean kits and the 2-category of stable functors

## Kits and negation

We have deliberately introduced a general notion of kit to emphasise its fundamental role in controlling stabilizers and quotients. In practice, however, one need impose conditions on kits to ensure desirable structure in the induced categories of stable presheaves (see Proposition 15). This may be done in several ways and we concentrate here on a principled approach based on a notion of *logical negation* described next.

▶ **Proposition 12.** *Let $\mathcal{A}$ be a kit on a groupoid $\mathbb{A}$. The following definition, for $a \in \mathbb{A}$,*

$$\mathcal{A}^{\perp}(a) = \{\, H \;\mid\; H \text{ is a subgroup of } !\mathbb{A}(u,u) \text{ satisfying: for all } G \in \mathcal{A}(a),\; G \cap H = \{\mathrm{id}_a\} \,\}$$

*yields a kit $\mathcal{A}^{\perp}$ called the* ***negation*** *(or* ***dual****) of $\mathcal{A}$.*

▶ **Definition 13.** *A* ***Boolean kit*** *is a kit $\mathcal{A}$ such that*

$$\mathcal{A}^{\perp\perp} = \mathcal{A}. \tag{8}$$

Boolean kits have useful closure properties:

▶ **Lemma 14.** *Let $\mathcal{A}$ be a Boolean kit on a groupoid $\mathbb{A}$. For every $a \in \mathbb{A}$, the set $\mathcal{A}(a)$ is closed under subgroups and under directed unions.*

From here, one can show that the category of stable presheaves induced by a Boolean kit has a rich structure inherited from the presheaf category.

▶ **Proposition 15.** *Let $\mathcal{A}$ be a kit on a groupoid $\mathbb{A}$. The category $\mathcal{S}(\mathbb{A},\mathcal{A})$ is closed under isomorphisms and coproducts taken in $\mathcal{P}(\mathbb{A})$. If the kit $\mathcal{A}$ is Boolean, then $\mathcal{S}(\mathbb{A},\mathcal{A})$ additionally inherits filtered colimits and all nonempty limits from $\mathcal{P}(\mathbb{A})$. Furthermore, the terminal presheaf is in $\mathcal{S}(\mathbb{A},\mathcal{A})$ if and only if $\mathcal{S}(\mathbb{A},\mathcal{A}) = \mathcal{P}(\mathbb{A})$.*

## The 2-category of Boolean kits and stable functors

The conditions defining stable functors (Definition 10) are preserved under composition, and identity functors are stable. Just like for polynomial endofunctors on sets, we will consider *cartesian* natural transformations (Definition 1) between stable functors.

▶ **Proposition 16.** *The following data forms a 2-category, called* **Stable***:*

- objects $(\mathbb{A},\mathcal{A})$*: groupoids with Boolean kits.*
- 1-cells $(\mathbb{A},\mathcal{A}) \to (\mathbb{B},\mathcal{B})$*: stable functors $\mathcal{S}(\mathbb{A},\mathcal{A}) \longrightarrow \mathcal{S}(\mathbb{B},\mathcal{B})$.*
- 2-cells*: cartesian natural transformations.*

*Compositions and identities in* **Stable** *are defined as for functors and natural transformations.*

The rest of the paper is devoted to the development and study of a bicategory of *stable species* of structures (Proposition 21) that provides an equivalent combinatorial view of **Stable**. The ultimate objective is our main theorem:

▶ **Theorem 17.** **Stable** *is a cartesian closed bicategory.*

Note the distinction between 2-categories, in which morphisms compose strictly, and bicategories, in which there are structural 2-cells in place of associativity and identity laws. This terminology extends to cartesian closed structure: **Stable** is cartesian as a 2-category but cartesian closed as a bicategory, since currying and uncurrying are up to isomorphism.

# 6 The cartesian closed bicategory of stable species

As a path to Theorem 17, this section introduces *stable species*: a combinatorial presentation of stable functors. Our methodology relies on generalised species [16], which we will enrich with kits.

## The bicategory of generalised species

A *generalised species* [18] from $\mathbb{A}$ to $\mathbb{B}$ is a functor $\mathbb{B}^{\mathrm{op}} \times !\mathbb{A} \longrightarrow \mathbf{Set}$ (or, equivalently, a functor $!\mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}\mathbb{B}$) where $!\mathbb{A}$ is the *symmetric strict monoidal completion* of $\mathbb{A}$. The construction of $!\mathbb{A}$ generalises the groupoid $\mathbf{B}$, which arises as $\mathbf{!1}$, and is motivated by the desire of passing from linear to cartesian higher-order structure.

The objects of $!\mathbb{A}$ are finite sequences $\langle a_1, \ldots, a_n \rangle$ $(n \in \mathbb{N})$ of objects of $\mathbb{A}$ and a morphism

$$\alpha : \langle a_1, \ldots, a_m \rangle \longrightarrow \langle b_1, \ldots b_n \rangle$$

consists of a pair $\big(\underline{\alpha}, (\alpha_i)_{i \in [n]}\big)$ where $\underline{\alpha} \in \mathbf{B}(m, n)$ and $\alpha_i : a_i \to b_{\underline{\alpha}(i)}$ is a morphism in $\mathbb{A}$ for every $i \in [m]$. The concatenation of sequences gives a monoidal tensor $(u, v) \longmapsto u \otimes v$ for $!\mathbb{A}$ having the empty list as monoidal unit.

It is helpful to understand the bicategory of generalised species in terms of *profunctors* (alternatively, *bimodules* or *distributors*) [50, 6, 7, 37]. A profunctor from $\mathbb{A}$ to $\mathbb{B}$, denoted $\mathbb{A} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{B}$, is a functor $\mathbb{B}^{\mathrm{op}} \times \mathbb{A} \longrightarrow \mathbf{Set}$. Small categories, profunctors, and natural transformations form a symmetric monoidal (compact) closed bicategory $\mathbf{Prof}$ with tensor product $\times$ and internal hom $\mathbb{A} \multimap \mathbb{B} := \mathbb{A}^{\mathrm{op}} \times \mathbb{B}$.

A generalised species from $\mathbb{A}$ to $\mathbb{B}$ is then simply a profunctor $!\mathbb{A} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{B}$ and, in fact, the bicategory of species is a coKleisli bicategory for ! as a pseudo-comonad on $\mathbf{Prof}$ [18, 13]. The identity species $\mathrm{id}_{\mathbb{A}} : !\mathbb{A} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{A}$ is the functor mapping a pair $(a, u) \in \mathbb{A}^{\mathrm{op}} \times !\mathbb{A}$ to the set

$$!\mathbb{A}(\langle a \rangle, u) \;\cong\; \begin{cases} \mathbb{A}(a, a') & \text{, if } u = \langle a' \rangle \\ \varnothing & \text{, otherwise} \end{cases}$$

The composition of species $F : !\mathbb{A} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{B}$ and $G : !\mathbb{B} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{C}$ is the species $G \circ F : !\mathbb{A} \longrightarrow\!\!\!\!\!\!\!+ \ \mathbb{C}$ that maps a pair $(c, u) \in \mathbb{C}^{\mathrm{op}} \times !\mathbb{A}$ to the set

$$\int^{v = \langle b_1, \ldots, b_n \rangle \in !\mathbb{B}} G(c, v) \times \int^{u_1, \ldots, u_n \in !\mathbb{A}} \prod_{i=1}^{n} F(b_i, u_i) \times !\mathbb{A}(u_1 \otimes \cdots \otimes u_n, u)$$

We call $\mathbf{Esp}$ the bicategory of groupoids, generalised species, and natural transformations. This is the restriction to groupoids of the bicategory of generalised species of structures defined in [18], whose objects can be arbitrary small categories.

## Stable species of structures

We now introduce a new bicategory $\mathbf{SEsp}$ (Proposition 21) whose objects are groupoids with kits and whose morphisms are generalised species with action restricted by the kits; we call these *stable species*.

We start by extending the ! construction to kits, so that for every groupoid with kit $(\mathbb{A}, \mathcal{A})$ we can set $!(\mathbb{A}, \mathcal{A}) \stackrel{\text{def}}{=} (!\mathbb{A}, !\mathcal{A})$. To define $!\mathcal{A}(u)$ for an object $u = \langle a_1, \ldots, a_n \rangle \in !\mathbb{A}$, we need a preliminary definition. Recall that an endomorphism $\alpha$ on $u$ in $!\mathbb{A}$ is a pair consisting of a permutation $\underline{\alpha} \in \mathfrak{S}_n$ and a sequence $(\alpha_i : a_i \to a_{\underline{\alpha}(i)})_{i \in [n]}$ of morphisms in $\mathbb{A}$. For every $i \in [n]$, define the endomorphism $\mathrm{loop}_i^{\alpha}$ on $a_i$ in $\mathbb{A}$ as the composite

$$a_i \xrightarrow{\alpha_i} a_{\underline{\alpha}(i)} \xrightarrow{\alpha_{\underline{\alpha}(i)}} a_{\underline{\alpha}^2(i)} \longrightarrow \cdots \longrightarrow a_{\underline{\alpha}^{o(i)-1}(i)} \xrightarrow{\alpha_{\underline{\alpha}^{o(i)-1}(i)}} a_{\underline{\alpha}^{o(i)}(i)} = a_i$$

where $o(i)$ is the smallest positive integer such that $\underline{\alpha}^{o(i)}(i) = i$. Equivalently, $o(i)$ is the length of the cycle containing $i$ in the disjoint cycle decomposition of the permutation $\underline{\alpha}$.

▶ **Definition 18.** *Let $(\mathbb{A}, \mathcal{A})$ be a groupoid with a kit. For $u = \langle a_1, \ldots, a_n \rangle \in !\mathbb{A}$, we define*

$$!\mathcal{A}(u) \stackrel{def}{=} \{\, H \mid H \textit{ is a subgroup of } !\mathbb{A}(u,u) \textit{ satisfying } \forall\, \alpha \in H.\, \forall\, i \in [n].\, \mathrm{loop}_i^\alpha \in \bigcup \mathcal{A}(a_i) \,\}^{\perp\perp}.$$

The closure under double dual directly ensures that we obtain a Boolean kit $!\mathcal{A} = \{\, !\mathcal{A}(u) \,\}_{u \in !\mathbb{A}}$ on $!\mathbb{A}$.

We now define stable species between groupoids with kits. As for the $\mathcal{K}$-species of Definition 4, our definition involves stabilizers controlled by kits. Note that, for a generalised species $F : !\mathbb{A} \longrightarrow \mathbb{B}$, the stabilizer $\mathrm{Stab}_F(p)$ of an element $p \in F(b, u)$ is a subgroup of $\mathbb{B}(b, b) \times !\mathbb{A}(u, u)$.

▶ **Definition 19.** *Let $(\mathbb{A}, \mathcal{A})$ and $(\mathbb{B}, \mathcal{B})$ be groupoids with kits. A* ***stable species*** *from $(\mathbb{A}, \mathcal{A})$ to $(\mathbb{B}, \mathcal{B})$ is a generalised species $F : !\mathbb{A} \longrightarrow \mathbb{B}$ such that, for every $b \in \mathbb{B}$, $u \in !\mathbb{A}$ and $p \in F(b, u)$, if $(\beta, \alpha) \in \mathrm{Stab}_F(p)$ then:*

$$\alpha \in \bigcup !\mathcal{A}(u) \Rightarrow \beta \in \bigcup \mathcal{B}(b) \quad \textit{and} \quad \beta \in \bigcup \mathcal{B}^\perp(b) \Rightarrow \alpha \in \bigcup (!\mathcal{A})^\perp(u). \tag{9}$$

In special cases we recover previous concepts:

**Stable presheaves.** The initial groupoid $\mathbf{0}$ has a unique, empty, Boolean kit. Moreover, $!\mathbf{0}$ is the terminal groupoid, which also admits a unique Boolean kit. It follows that stable species from $(\mathbf{0}, \varnothing)$ to any $(\mathbb{A}, \mathcal{A})$ correspond to presheaves in $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$.

**Free Joyal species.** Let $\mathcal{K}_1$ be the unique Boolean kit on the terminal groupoid $\mathbf{1}$. Unfolding Definition 18, we get that $!\mathcal{K}_1$ is the maximal kit (consisting of all subgroups) on $!\mathbf{1} = \mathbf{B}$ with dual kit $(!\mathcal{K}_1)^\perp$ the minimal Boolean kit (consisting of trivial subgroups). Then, the stable species from $(\mathbf{1}, \mathcal{K}_1)$ to $(\mathbf{1}, \mathcal{K}_1)$ coincide with the free species $\mathbf{B} \longrightarrow \mathbf{Set}$.

More generally, by considering pairs of endomorphisms of the form $(\beta, \mathrm{id}_u)$ and $(\mathrm{id}_b, \alpha)$ in the two implications of (9) above, and observing that Boolean kits always contain identities, we obtain the following two properties:

▶ **Lemma 20.** *Let $F$ be a stable species from $(\mathbb{A}, \mathcal{A})$ to $(\mathbb{B}, \mathcal{B})$ with $\mathcal{A}$ and $\mathcal{B}$ Boolean.*

***($\mathcal{B}$-stability)*** *The corresponding functor $F : !\mathbb{A} \to \boldsymbol{\mathcal{P}}(\mathbb{B})$ factors through the inclusion $\boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B}) \hookrightarrow \boldsymbol{\mathcal{P}}(\mathbb{B})$.*

***($!\mathcal{A}$-freeness)*** *For $\alpha \in !\mathbb{A}(u, u)$, if $F\alpha \in \boldsymbol{\mathcal{P}}\mathbb{B}(Fu, Fu)$ fixes an element of the presheaf $Fu$, then $\alpha \in (!\mathcal{A})^\perp(u)$.*

One verifies the following directly:

▶ **Proposition 21.** *The following data forms a bicategory, called* **SEsp***:*

- objects $(\mathbb{A}, \mathcal{A})$*: groupoids with Boolean kits;*
- 1-cells $(\mathbb{A}, \mathcal{A}) \to (\mathbb{B}, \mathcal{B})$*: stable species* $!(\mathbb{A}, \mathcal{A}) \longrightarrow (\mathbb{B}, \mathcal{B})$*;*
- 2-cells*: natural transformations.*

*Compositions and identities in* **SEsp** *are defined as for generalised species.*

### Cartesian closed structure in the bicategory of stable species

The cartesian closed categorical structure of **SEsp** extends that in **Esp** without difficulty:

**Cartesian products.** The bicategory **Esp** has finite products given by the finite sum of groupoids. For a finite family of groupoids with kits $\{ (\mathbb{A}_i, \mathcal{A}_i) \}_{i\in[n]}$, the sum groupoid $\mathbb{A}_1 + \cdots + \mathbb{A}_n$ has boolean kit $\mathcal{A}_1 + \cdots + \mathcal{A}_n$ defined as $(\mathcal{A}_1 + \cdots + \mathcal{A}_n)(\imath_i(a)) = \mathcal{A}_i(a)$ where $\imath_i$ is the coproduct inclusion. This contruction endows **SEsp** with a finite product structure.

**Higher-order structure.** We recall the situation in **Esp** [18]. For groupoids $\mathbb{A}$ and $\mathbb{B}$, the function space $\mathbb{A} \Rightarrow \mathbb{B}$ is defined as $!\mathbb{A} \multimap \mathbb{B}$. The proof of cartesian closure relies on a fundamental canonical equivalence

$$!\mathbb{A} \times !\mathbb{B} \simeq !(\mathbb{A} + \mathbb{B})$$

given by the composite

$$!\mathbb{A} \times !\mathbb{B} \xrightarrow{!(\imath_1)\times!(\imath_2)} !(\mathbb{A} + \mathbb{B}) \times !(\mathbb{A} + \mathbb{B}) \xrightarrow{\otimes} !(\mathbb{A} + \mathbb{B})$$

and the symmetric monoidal extension of the functor

$$\mathbb{A} + \mathbb{B} \to !\mathbb{A} \times !\mathbb{B} : \begin{cases} \imath_1(a) \longmapsto (\langle a \rangle, \varnothing) & \text{, for } a \in \mathbb{A} \\ \imath_2(b) \longmapsto (\varnothing, \langle b \rangle) & \text{, for } b \in \mathbb{B} \end{cases}$$

With this equivalence, we have a chain of equivalences of hom-categories as follows:

$$\mathbf{Esp}(\mathbb{C}, \mathbb{A} \Rightarrow \mathbb{B}) = \mathbf{Prof}(!\mathbb{C}, !\mathbb{A} \multimap \mathbb{B}) \cong \mathbf{Prof}(!\mathbb{C} \times !\mathbb{A}, \mathbb{B}) \simeq \mathbf{Prof}(!(\mathbb{C} + \mathbb{A}), \mathbb{B}) = \mathbf{Esp}(\mathbb{C} + \mathbb{A}, \mathbb{B})$$

This provides the required *currying/uncurrying* pseudo-natural equivalence, which we must extend to **SEsp**. This we achieve by setting $(\mathbb{A}, \mathcal{A}) \Rightarrow (\mathbb{B}, \mathcal{B}) = (\mathbb{A} \Rightarrow \mathbb{B}, \mathcal{A} \Rightarrow \mathcal{B})$ where $(\mathcal{A} \Rightarrow \mathcal{B})(u, b)$ consists of all the subgroups $H$ of $!\mathbb{A}(u, u) \times \mathbb{B}(b, b)$ such that (9) is satisfied for every $(\alpha, \beta) \in H$.

▶ **Theorem 22.** *The bicategory* **SEsp** *is cartesian closed.*

▶ Remark. To establish the theorem in full rigour we first develop a theory of profunctors underlying stable species. This relies on a notion of *stabilised profunctor* between groupoids with Boolean kits and on the fact that the ! construction is a linear exponential pseudo-comonad over an associated $\star$-autonomous bicategory **SProf**. This is a fairly technical development but along expected lines, given the linear exponential pseudo-comonad structure of ! on **Prof** [18, 13] and our construction of Boolean kits based on negation. This will be developed fully in a companion paper, exploring in depth the connections with linear logic and linear negation.

## 7 The biequivalence between stable species and stable functors

We show that stable species and stable functors are alternative presentations of the same model. Formally, we establish a bicategorical equivalence **SEsp** $\simeq$ **Stable** (Theorem 32). From stable species to stable functors, we study the class of analytic functors induced by stable species and show that they are stable. In the opposite direction, we show how to recover the coefficients of a stable species from a stable functor in terms of generic factorisations.

## 7.1 From stable species to stable functors

The *analytic functor* induced by a generalised species $P : !\mathbb{A} \longrightarrow\!\!\!\!\!\!+\;\, \mathbb{B}$ is the functor $\boldsymbol{\mathcal{P}}(\mathbb{A}) \longrightarrow \boldsymbol{\mathcal{P}}(\mathbb{B})$ that maps a presheaf $X \in \boldsymbol{\mathcal{P}}(\mathbb{A})$ to the presheaf on $\mathbb{B}$ defined, for $b \in \mathbb{B}$, as

$$b \longmapsto \int^{u=\langle a_1,\dots,a_n\rangle \in !\mathbb{A}} P(b,u) \times \prod_{i=1}^{n} X(a_i) \tag{10}$$

This formula, and indeed the earlier one for Joyal species (4), are obtained through the universal construction of *left Kan extension*:

▶ **Definition 23.** *For groupoids $\mathbb{A}, \mathbb{B}$, and a generalised species $P : !\mathbb{A} \longrightarrow\!\!\!\!\!\!+\;\, \mathbb{B}$, the analytic functor described pointwise above is a left Kan extension of $P$ along the functor $s_{\mathbb{A}} : !\mathbb{A} \longrightarrow \boldsymbol{\mathcal{P}}(\mathbb{A}) : \langle a_1,\dots,a_n\rangle \longmapsto \sum_{i=1}^{n} \mathrm{y}(a_i)$, as on the left below:*

*For $\mathbb{A} = \mathbb{B} = \mathbf{1}$ and $P$ viewed as a species $\mathbf{B} \longrightarrow \mathbf{Set}$, this corresponds to the diagram on the right above, where $j$ is the inclusion functor, and induces the formula* (4)*.*

Given a stable species from $(\mathbb{A}, \mathcal{A})$ to $(\mathbb{B}, \mathcal{B})$, one can apply the formula (10) to obtain a stable functor $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) \longrightarrow \boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B})$.

▶ **Proposition 24.** *Let $P : !(\mathbb{A}, \mathcal{A}) \longrightarrow\!\!\!\!\!\!+\;\, (\mathbb{B}, \mathcal{B})$ in* **SEsp***. The restricted left Kan extension $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) \hookrightarrow \boldsymbol{\mathcal{P}}(\mathbb{A}) \xrightarrow{\mathrm{Lan}_{s_{\mathbb{A}}} P} \boldsymbol{\mathcal{P}}(\mathbb{B})$ factors through the inclusion $\boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B}) \hookrightarrow \boldsymbol{\mathcal{P}}(\mathbb{B})$. Furthermore, the resulting functor $\widetilde{P} : \boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) \longrightarrow \boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B})$ is stable.*

To extend the mapping $\widetilde{(-)}$ to a functor $\mathbf{SEsp}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big) \longrightarrow \mathbf{Stable}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$, we verify that natural transformations between stable species are mapped to cartesian natural transformations between stable functors:

▶ **Proposition 25.** *Let $(\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})$ be groupoids with Boolean kits and let $f : P \to Q$ be a natural transformation between stable species $P, Q : !(\mathbb{A}, \mathcal{A}) \longrightarrow\!\!\!\!\!\!+\;\, (\mathbb{B}, \mathcal{B})$. The natural transformation $\widetilde{f} : \widetilde{P} \to \widetilde{Q} : \boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) \longrightarrow \boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B})$, canonically induced by left Kan extension, is cartesian.*

## 7.2 From stable functors to stable species

We have given the components of a pseudo-functor $\widetilde{(-)} : \mathbf{SEsp} \longrightarrow \mathbf{Stable}$. We proceed to show that it is part of a biequivalence, and construct a pseudo-inverse $\mathbf{Stable} \longrightarrow \mathbf{SEsp}$. We rely on a characterisation of stable functors in terms of *generic morphisms*:

▶ **Definition 26.** *Let $T : \mathscr{C} \to \mathscr{D}$ be a functor between categories $\mathscr{C}$ and $\mathscr{D}$. A morphism $g : d \to T(c)$ in $\mathscr{D}$ is called* **generic** *if, for every commuting square as on the left below*

$$\begin{array}{ccc} & T(z) & \\ {\scriptstyle T(f)}\nearrow & & \nwarrow{\scriptstyle T(f')} \\ T(c) & & T(c') \\ {\scriptstyle g}\nwarrow & & \nearrow{\scriptstyle g'} \\ & d & \end{array} \qquad \begin{array}{ccc} & z & \\ {\scriptstyle f}\nearrow & & \nwarrow{\scriptstyle f'} \\ c & \xrightarrow{k} & c' \\ T(c) & \xrightarrow{T(k)} & T(c') \\ {\scriptstyle g}\nwarrow & & \nearrow{\scriptstyle g'} \\ & d & \end{array}$$

*there exists a unique morphism $k : c \to c'$ in $\mathscr{C}$ making the two triangles on the right above commute. The functor $T$ is said to* ***admit generic factorisations*** *if every morphism $h : d \to T(z)$ has a factorisation $h = T(f) \circ g$ for some $f : c \to z$ and $g : d \to T(c)$ with $g$ generic.*

Generic morphisms (elsewhere known as *candidates* [45] or *strict generic* [49], and a strict version of the *generic elements* considered in [32, 16]) provide the elements in the construction of a stable species from a stable functor. They also correspond to the *normal forms* studied in [26, 28]. The presence of generic factorisations is closely related to the existence of local left adjoints, and we have the following:

▶ **Proposition 27.** *Let $(\mathbb{A}, \mathcal{A})$ and $(\mathbb{B}, \mathcal{B})$ be groupoids with Boolean kits. A functor from $\boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A})$ to $\boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B})$ is stable if and only if admits generic factorisations, is finitary, and preserves regular epimorphisms.*

We use this to show that the following operation provides a pseudo-inverse to $P \longmapsto \widetilde{P}$:

▶ **Definition 28.** *For $(\mathbb{A}, \mathcal{A})$ and $(\mathbb{B}, \mathcal{B})$ groupoids with Boolean kits, the* ***trace*** *of a functor $T : \boldsymbol{\mathcal{S}}(\mathbb{A}, \mathcal{A}) \longrightarrow \boldsymbol{\mathcal{S}}(\mathbb{B}, \mathcal{B})$ is the generalised species $\mathrm{Tr}(T) : !\mathbb{A} \longrightarrow\!\!\!\!\!+ \mathbb{B}$ with object mapping*

$$(b, u) \longmapsto \{\, g : \mathrm{y}_{\mathbb{B}}(b) \to T(s_{\mathbb{A}} u) \text{ in } \boldsymbol{\mathcal{P}}(\mathbb{B}) \mid g \text{ is generic} \,\}$$

*and functorial action given by composition (which is well-defined because genericity is invariant under isomorphism).*

▶ **Proposition 29.** *For a functor $T$ in $\mathbf{Stable}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$, the species $\mathrm{Tr}(T)$ is in $\mathbf{SEsp}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$.*

Cartesian natural transformations preserve and reflect generic morphisms [49]. Thus, one can immediately extend the trace operation to 2-cells:

▶ **Proposition 30.** *For a cartesian natural transformation $f : S \to T$ in $\mathbf{Stable}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$, the mapping*

$$\mathrm{Tr}(f)_{(b,u)} : \big(\, g : \mathrm{y}_{\mathbb{B}}(b) \to S(s_{\mathbb{A}} u) \,\big) \longmapsto \big(\, f_{s_{\mathbb{A}}(u)} \circ g : \mathrm{y}_{\mathbb{B}}(b) \to T(s_{\mathbb{A}} u) \,\big) \qquad (b \in \mathbb{B}, u \in !\mathbb{A})$$

*provides the components of a natural transformation $\mathrm{Tr}(f) : \mathrm{Tr}(S) \to \mathrm{Tr}(T)$.*

It remains to exhibit local equivalences between the corresponding hom-categories:

▶ **Lemma 31.**

1. *For a stable species $F \in \mathbf{SEsp}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$, $\mathrm{Tr}(\widetilde{F}) \cong F$.*
2. *For a stable functor $S \in \mathbf{Stable}\big((\mathbb{A}, \mathcal{A}), (\mathbb{B}, \mathcal{B})\big)$, $\widetilde{\mathrm{Tr}(S)} \cong S$.*

▶ **Theorem 32.** *There is a biequivalence of bicategories $\mathbf{SEsp} \simeq \mathbf{Stable}$.*

As biequivalences preserve cartesian closed structure, our main Theorem 17 is a corollary of Theorem 22.

# 8 Conclusion

We have defined a new cartesian closed bicategorical model that we have presented independently in combinatorial and extensional forms, respectively as:

- stable species of structures between groupoids with Boolean kits, and
- stable functors between categories of stable presheaves over groupoids with Boolean kits.

Restricting our extensional model to discrete groupoids, one precisely obtains finitary polynomial functors $\mathbf{Set}^I \to \mathbf{Set}^J$ between categories of indexed sets, corresponding to Girard's *normal functors* [26]. Polynomial functors of this type (finitary or not) are typically represented in combinatorial form as *polynomials* in **Set**; namely, diagrams $(I \leftarrow E \to B \to J)$ of sets and functions representing operators with sorted (or coloured) arities (§2) [22, 4]. In the finitary case it is not hard to translate between these and our stable species representation.

However, extending the above to incorporate higher-order structure seems to require moving on to the general context of groupoids: even when $I$ and $J$ are discrete groupoids, the function space $I \Rightarrow J = !I^{\mathrm{op}} \times J$ is *not* discrete, and to remain within polynomials one must make explicit and control the groupoid action.

In recent work, Finster, Lucas, Mimram and Seiller [12] present another groupoid model, which they describe in the language of homotopy type theory. The relationship with our model should be considered, also in connection to the work of Kock et al. [33, 23].

### Connections with stable domain theory

Our model provides a form of generalised domain theory in which continuous functions between domains are generalised to finitary functors between domain-like categories (see e.g. [3]); this fits in the general research programme outlined in [29]. Specifically, we have a generalised form of *stable* domain theory in the sense of Berry [9] and Girard [24]; stable functions are finitary local right adjoints between stable domains and Berry's stable order amounts to unique degenerate cartesian natural transformations.

In our endeavour, we follow Lamarche [35] and Taylor [45, 46], who in the 1980s pioneered the categorification of stable domain theory. Taylor's work is especially relevant: he introduced *creeds*, a combinatorial structure on groupoids used to control actions at higher order, and even raised the thought of a connection with Joyal's ideas [45, page 172]. Whilst our Boolean kits are rather different from creeds, the present work recasts these ideas in a modern structural combinatorial setting and bicategorical language, suggesting new avenues for research.

### Further work on bicategorical models of linear logic and differentiation

The bicategory **SEsp** is obtained from a bicategorical model of classical linear logic **SProf**, whose theory we will present in a future paper. A question we are investigating is whether this model can be obtained through a bicategorical glueing construction by means of an orthogonality technique [30].

Another promising direction is the study of formal differentiation. In this respect, there are likely connections with several lines of work, including: differentiation for polynomial functors in type theory [39, 2, 27, 15]; differentiation for analytic functors in combinatorics [8, 34]; and differential linear logic [11, 14, 17, 10] of which **SProf** is a bicategorical model.

For combinatorial species, formal differentiation gives rise to formal integration and to the study of differential equations from that perspective [38]. Free species, which can always be integrated [34], are most useful in that context; our bicategory **SEsp** is a promising setting for extending these notions to a higher-order logical setting.

## References


1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

2 Michael Abbott, Thorsten Altenkirch, Conor McBride, and Neil Ghani. $\partial$ for data: Differentiating data structures. *Fundam. Informaticae*, 65(1-2):1–28, 2005.

3 Jiří Adámek. A categorical generalization of Scott domains. *Mathematical Structures in Computer Science*, 7(5):419–443, 1997. doi:10.1017/S0960129597002351.

4 Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015. doi:10.1017/S095679681500009X.

5 Steve Awodey and Clive Newstead. Polynomial pseudomonads and dependent type theory, 2018. doi:10.48550/ARXIV.1802.00997.

6 Jean Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*, pages 1–77, Berlin, Heidelberg, 1967. Springer Berlin Heidelberg.

7 Jean Bénabou. Distributors at work, 2000. Lecture notes written by Thomas Streicher. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/FIBR/DiWo.pdf.

8 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.

9 Gérard Berry. Stable models of typed lambda-calculi. In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 72–89, Berlin, Heidelberg, 1978. Springer-Verlag.

10 Thomas Ehrhard. An introduction to differential linear logic: Proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28:995–1060, 2018. doi:10.1017/S0960129516000372.

11 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

12 Eric Finster, Samuel Mimram, Maxime Lucas, and Thomas Seiller. A cartesian bicategory of polynomial functors in homotopy type theory. In Ana Sokolova, editor, *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*, volume 351 of *EPTCS*, pages 67–83, 2021. doi:10.4204/EPTCS.351.5.

13 M. Fiore, N. Gambino, M. Hyland, and G. Winskel. Relative pseudomonads, Kleisli bicategories, and substitution monoidal structures. *Selecta Mathematica*, 24(3):2791–2830, 2018. doi:10.1007/s00029-017-0361-3.

14 Marcelo Fiore. Mathematical models of computational and combinatorial structures. In *International Conference on Foundations of Software Science and Computation Structures*, pages 25–46. Springer, 2005.

15 Marcelo Fiore. Discrete generalised polynomial functors. In *Automata, Languages, and Programming*, pages 214–226. Springer, 2012.

16 Marcelo Fiore. Analytic functors between presheaf categories over groupoids. *Theor. Comput. Sci.*, 546:120–131, 2014. doi:10.1016/j.tcs.2014.03.004.

17 Marcelo Fiore. An axiomatics and a combinatorial model of creation/annihilation operators, 2015. doi:10.48550/ARXIV.1506.06402.

18 Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. The cartesian closed bicategory of generalised species of structures. *J. Lond. Math. Soc. (2)*, 77(1):203–220, 2008. doi:10.1112/jlms/jdm096.

19 Marcelo Fiore and Philip Saville. A type theory for cartesian closed bicategories. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

20 Zeinab Galal. A bicategorical model for finite nondeterminism. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

21 Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003. doi:10.1007/978-3-540-24849-1_14.

22 Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013. doi:10.1017/S0305004112000394.

23 David Gepner, Rune Haugseng, and Joachim Kock. $\infty$-Operads as analytic monads. *International Mathematics Research Notices*, 2021. doi:10.1093/imrn/rnaa332.

24 Jean-Yves Girard. The system F of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2):159–192, 1986. doi:10.1016/0304-3975(86)90044-7.

25 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

26 Jean-Yves Girard. Normal functors, power series and $\lambda$-calculus. *Ann. Pure Appl. Logic*, 37(2):129–177, 1988. doi:10.1016/0168-0072(88)90025-5.

27 Makoto Hamana and Marcelo Fiore. A foundation for gadts and inductive families: Dependent polynomial functor approach. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pages 59–70. Association for Computing Machinery, 2011. doi:10.1145/2036918.2036927.

28 Ryu Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1):113–175, 2002. doi:10.1016/S0304-3975(00)00349-2.

29 Martin Hyland. Some reasons for generalising domain theory. *Math. Struct. Comput. Sci.*, 20(2):239–265, 2010. doi:10.1017/S0960129509990375.

30 Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theoretical Computer Science*, 294(1):183–231, 2003. doi:10.1016/S0304-3975(01)00241-9.

31 André Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42(1):1–82, 1981. doi:10.1016/0001-8708(81)90052-9.

32 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.

33 Joachim Kock. Data types with symmetries and polynomial functors over groupoids. *Electronic Notes in Theoretical Computer Science*, 286:351–365, 2012. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII). doi:10.1016/j.entcs.2013.01.001.

34 Gilbert Labelle. Combinatorial integration (Part I, Part II). *ACM Commun. Comput. Algebra*, 49(1):35, June 2015. doi:10.1145/2768577.2768653.

35 François Lamarche. *Modelling polymorphism with categories.* PhD thesis, McGill University, 1988.

36 François Lamarche. Quantitative domains and infinitary algebras. *Theoretical Computer Science*, 94:37–62, 1992.

37 F. William Lawvere. Metric spaces, generalized logic, and closed categories. *Rendiconti del Seminario Matematico e Fisico di Milano*, 43:135–166, 1973. Republished in: *Reprints in Theory and Applications of Categories*, No. 1 (2002) pp 1–37.

38 P. Leroux and G.X. Viennot. Combinatorial resolution of systems of differential equations. IV. Separation of variables. *Discrete Mathematics*, 72(1-3):237–250, 1988. doi:10.1016/0012-365X(88)90213-0.

39 Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001. Unpublished manuscript.

40 Paul-André Melliès. Template games and differential linear logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

**41** Lê Thành Dung Nguyên and Pierre Pradic. From normal functors to logarithmic space queries. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 123:1–123:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.123.

**42** Federico Olimpieri. Intersection type distributors. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15. IEEE, 2021.

**43** David Spivak. Poly: An abundant categorical setting for mode-dependent dynamics, 2020. doi:10.48550/ARXIV.2005.01894.

**44** Ross Street. The petit topos of globular sets. *Journal of Pure and Applied Algebra*, 154(1):299–315, 2000. doi:10.1016/S0022-4049(99)00183-8.

**45** Paul Taylor. Quantitative domains, groupoids and linear logic. In *Category Theory and Computer Science*, pages 155–181. Springer, 1989.

**46** Paul Taylor. An algebraic approach to stable domains. *Journal of Pure and Applied Algebra*, 64(2):171–203, 1990. doi:10.1016/0022-4049(90)90156-C.

**47** Takeshi Tsukada, Kazuyuki Asada, and C.-H. Luke Ong. Generalised species of rigid resource terms. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017.

**48** Tamara Von Glehn. *Polynomials and models of type theory.* PhD thesis, University of Cambridge, 2015.

**49** Mark Weber. Generic morphisms, parametric representations and weakly cartesian monads. *Theory and Applications of Categories*, 13(14):191–234, 2004.

**50** Nobuo Yoneda. On Ext and exact sequences. *Journal of the Faculty of Science, Imperial University of Tokyo*, 8:507–576, 1960.

# Galois Connecting Call-by-Value and Call-by-Name

**Dylan McDermott**
Reykjavik University, Iceland

**Alan Mycroft**
University of Cambridge, UK

## Abstract

We establish a general framework for reasoning about the relationship between call-by-value and call-by-name.

In languages with side-effects, call-by-value and call-by-name executions of programs often have different, but related, observable behaviours. For example, if a program might diverge but otherwise has no side-effects, then whenever it terminates under call-by-value, it terminates with the same result under call-by-name. We propose a technique for stating and proving these properties. The key ingredient is Levy's call-by-push-value calculus, which we use as a framework for reasoning about evaluation orders. We construct maps between the call-by-value and call-by-name interpretations of types. We then identify properties of side-effects that imply these maps form a Galois connection. These properties hold for some side-effects (such as divergence), but not others (such as mutable state). This gives rise to a general reasoning principle that relates call-by-value and call-by-name. We apply the reasoning principle to example side-effects including divergence and nondeterminism.

**2012 ACM Subject Classification** Theory of computation → Semantics and reasoning; Theory of computation → Denotational semantics; Theory of computation → Categorical semantics

**Keywords and phrases** computational effect, evaluation order, call-by-push-value, categorical semantics

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2022.32

**Related Version** *Extended Version*: `https://arxiv.org/abs/2202.08246` [16]

**Funding** *Dylan McDermott*: Supported by an EPSRC studentship, and by Icelandic Research Fund project grant no. 196323-053.

## 1 Introduction

Suppose that we have a language in which terms can be statically tagged either as using call-by-value evaluation or as using call-by-name evaluation. Each program in this language would therefore use a mix of call-by-value and call-by-name at runtime. Given any such program $M$, we can construct a new program $M'$ by changing call-by-value to call-by-name for some subterm. The question we consider in this paper is: what is the relationship between the observable behaviour of $M$ and the observable behaviour of $M'$?

For a language with side-effects (such as divergence), changing the evaluation order in this way will in general change the behaviour of the program, but for some side-effects we can often say something about how we expect the behaviour to change:

- If there are no side-effects at all (in particular, programs are normalizing), the choice of evaluation order is irrelevant: $M$ and $M'$ terminate with the same result.
- If there are diverging terms (for instance, via recursion), then the behaviour may change: a program might diverge under call-by-value and return a result under call-by-name. However, we can say something about how the behaviour changes: if $M$ terminates with some result, then $M'$ terminates with the same result.
- If nondeterminism is the only side-effect, every result of $M$ is a possible result of $M'$.

These three instances of the problem are intuitively obvious, and each can be proved separately. We develop a *general* technique for proving these properties.

© Dylan McDermott and Alan Mycroft;
licensed under Creative Commons License CC-BY 4.0
7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 32; pp. 32:1–32:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The idea is to use a calculus that captures both call-by-value and call-by-name, as a setting in which we can reason about both evaluation orders (this is where $M$ and $M'$ live). The calculus we use is Levy's *call-by-push-value* (CBPV) [11]. Levy describes how to translate (possibly open) expressions $e$ into CBPV terms $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$, which respectively correspond to call-by-value and call-by-name. We study the relationship between the behaviour of $(\!|e|\!)^{\mathrm{v}}$ and the behaviour of $(\!|e|\!)^{\mathrm{n}}$ in a given program context.

The main obstacle is that $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ have different types, and hence cannot be directly compared. Our solution to this is based on Reynolds's work relating direct and continuation semantics of the $\lambda$-calculus [25]: we identify maps between the call-by-value and call-by-name interpretations, and compose these with the translations of expressions to arrive at two terms that *can* be compared directly. We show that, under certain conditions (satisfied only for some side-effects, such as our examples), the maps between call-by-value and call-by-name form a *Galois connection* (Theorem 17). This fact gives rise to a general *reasoning principle* (Theorem 21) that we use to compare call-by-value with call-by-name. Given any preorder $\preccurlyeq$ that captures the property we wish to show about programs, our reasoning principle gives sufficiency conditions for showing $M \preccurlyeq M'$, where $M'$ is constructed as above by replacing call-by-value with call-by-name. We apply our reasoning principle to examples by choosing different relations $\preccurlyeq$; each of these relations indicates the extent to which changing evaluation order affects the behaviour of the program. In the divergence example $N \preccurlyeq N'$ is defined to mean termination of $N$ implies termination of $N'$ with the same result; in the other examples $\preccurlyeq$ similarly mirrors the properties described informally above.

Rather than just considering some fixed collection of (allowable) side-effects, we work abstractly and identify properties of side-effects that enable us to relate call-by-value and call-by-name. An advantage of our approach is that the properties can be derived by looking at the structure of the two maps between evaluation orders.

Our reasoning principle relies on the existence of some denotational model of the side-effects. We construct the Galois connections and relate the call-by-value and call-by-name translations inside the model itself. Crucially, we use *order-enriched* models, which order the denotations of terms. The ordering on denotations is necessary to obtain a general reasoning principle. (Our example properties cannot be proved by showing that denotations are equal, because they are not symmetric.) Working inside the semantics rather than using syntactic logical relations makes it easier to prove and to use our reasoning principle, especially for the divergence example.

In Section 2 we summarize the call-by-push-value calculus (CBPV) and the call-by-value and call-by-name translations. We then make the following contributions:

- We describe an *order-enriched* categorical semantics for CBPV (Section 3).
- We define maps between the call-by-value and call-by-name translations (Section 4), and show that they form a Galois connection for side-effects satisfying certain conditions (Theorem 17).
- We use the Galois connection to prove a novel reasoning principle (Theorem 21) that relates the call-by-value and call-by-name translations of expressions (Section 5).

Throughout, we consider three different examples: no side-effects, divergence, and non-determinism. We apply our reasoning principle to each, proving all of the above properties. Our motivation is partly to demonstrate the Galois connection technique as a way of reasoning about different semantics of a given language. Call-by-value and call-by-name is one example of this (and Reynolds's original application to direct and continuation semantics is another).

## 2 Call-by-push-value, call-by-value, and call-by-name

Levy [11, 13] introduced call-by-push-value (CBPV) as a calculus that captures both call-by-value and call-by-name. We reason about the relationship between call-by-value and call-by-name evaluation inside CBPV.

The syntax of CBPV terms is stratified into two kinds: *values* $V, W$ do not reduce, *computations* $M, N$ might reduce (possibly with side-effects). The syntax of types is similarly stratified into *value types* $A, B$ and *computation types* $\underline{C}, \underline{D}$.

$$\begin{array}{rrcl} \text{value types} & A, B & ::= & \mathbf{bool} \mid \mathbf{U}\,\underline{C} \\ \text{computation types} & \underline{C}, \underline{D} & ::= & A \to \underline{C} \mid \mathbf{F}\,A \\ \text{values} & V, W & ::= & x \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{thunk}\,M \\ \text{computations} & M, N & ::= & \lambda x{:}A.\,M \mid V`M \mid \mathbf{return}\,V \mid M\ \mathbf{to}\ x.\,N \\ & & & \mid \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \mathbf{force}\,V \end{array}$$

We include only a minimal subset of CBPV (containing higher-order functions, which are the main source of difficulty).

We include booleans (the value type **bool**) as a representative base type. The value type $\mathbf{U}\,\underline{C}$ is the type of *thunks* of computations of type $\underline{C}$. Elements of $\mathbf{U}\,\underline{C}$ are introduced using **thunk**: the value $\mathbf{thunk}\,M$ is the suspension of the computation term $M$. The corresponding eliminator is **force**, which is the inverse of **thunk**. Computation types include function types (where functions send values to computations). Function application is written $V`M$, where $V$ is the argument and $M$ is the function to apply. The *returner type* $\mathbf{F}\,A$ has as elements computations that return elements of the value type $A$; these computations may have side-effects. Elements of $\mathbf{F}\,A$ are introduced by **return**; the computation $\mathbf{return}\,V$ immediately returns the value $V$ (with no side-effects). Computations can be sequenced using $M\ \mathbf{to}\ x.\,N$. This first evaluates $M$ (which is required to have returner type), and then evaluates $N$ with $x$ bound to the result of $M$. (It is similar to `M >>= \x -> N` in Haskell.) The syntax we give here does not include any method of introducing effects; we extend CBPV with divergence (via recursion) and with nondeterminism in Section 2.2.

The evaluation order in CBPV is fixed for each program. The only primitive that causes the evaluation of two separate computations is **to**, which implements eager sequencing. Thunks give us more control over the evaluation order: they can be arbitrarily duplicated and discarded, and can be forced in any order chosen by the program. This is how CBPV captures both call-by-value and call-by-name.

CBPV has two typing judgments: $\Gamma \vdash V : A$ for values and $\Gamma \vdash_c M : \underline{C}$ for computations. *Typing contexts* $\Gamma$ are ordered lists of (variable, value type) pairs. We require that no variable appears more than once in any typing context. Figure 1 gives the typing rules. Rules that add a new variable to a typing context implicitly require that the variable is fresh. We write $\diamond$ for the empty typing context, $V : A$ as an abbreviation for $\diamond \vdash V : A$, and $M : \underline{C}$ as an abbreviation for $\diamond \vdash_c M : \underline{C}$.

We give an operational semantics for CBPV. This consists of a big-step evaluation relation $M \Downarrow R$, which means the computation $M$ evaluates to $R$. Here $R$ ranges over *terminal computations*, which are the subset of computations with an introduction form on the outside:

$$R \ ::= \ \lambda x{:}A.\,M \mid \mathbf{return}\,V$$

We only evaluate closed, well-typed computations, so when we write $M \Downarrow R$ we assume $M : \underline{C}$ for some $\underline{C}$ (this implies $R : \underline{C}$). Reduction therefore cannot get stuck. The rules defining $\Downarrow$ are given in Figure 2. All terminal computations evaluate to themselves. Since we have

$\boxed{\Gamma \vdash V : A}$

$$\frac{}{\Gamma \vdash x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{\Gamma \vdash_c M : \underline{C}}{\Gamma \vdash \mathbf{thunk}\, M : \mathbf{U}\underline{C}}$$

$\boxed{\Gamma \vdash_c M : \underline{C}}$

$$\frac{\Gamma, x : A \vdash_c M : \underline{C}}{\Gamma \vdash_c \lambda x{:}A.\, M : A \to \underline{C}} \qquad \frac{\Gamma \vdash V : A \quad \Gamma \vdash_c M : A \to \underline{C}}{\Gamma \vdash_c V`M : \underline{C}} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash_c \mathbf{return}\, V : \mathbf{F}\, A}$$

$$\frac{\Gamma \vdash_c M : \mathbf{F}\, A \quad \Gamma, x : A \vdash_c N : \underline{C}}{\Gamma \vdash_c M \ \mathbf{to}\ x.\, N : \underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{bool} \quad \Gamma \vdash_c M_1 : \underline{C} \quad \Gamma \vdash_c M_2 : \underline{C}}{\Gamma \vdash_c \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{U}\underline{C}}{\Gamma \vdash_c \mathbf{force}\, V : \underline{C}}$$

**Figure 1** CBPV typing rules.

$$\frac{}{\lambda x{:}A.\, M \Downarrow \lambda x{:}A.\, M} \qquad \frac{M \Downarrow \lambda x{:}A.\, N \qquad N[x \mapsto V] \Downarrow R}{V`M \Downarrow R}$$

$$\frac{}{\mathbf{return}\, V \Downarrow \mathbf{return}\, V} \qquad \frac{M \Downarrow \mathbf{return}\, V \qquad N[x \mapsto V] \Downarrow R}{M\ \mathbf{to}\ x.\, N \Downarrow R} \qquad \frac{M \Downarrow R}{\mathbf{force}\,(\mathbf{thunk}\, M) \Downarrow R}$$

$$\frac{M_1 \Downarrow R}{\mathbf{if\ true\ then}\ M_1\ \mathbf{else}\ M_2 \Downarrow R} \qquad \frac{M_2 \Downarrow R}{\mathbf{if\ false\ then}\ M_1\ \mathbf{else}\ M_2 \Downarrow R}$$

**Figure 2** Big-step operational semantics of CBPV.

not yet included any way of forming impure computations, the semantics is deterministic and normalizing: given any $M : \underline{C}$, there is exactly one terminal computation $R$ such that $M \Downarrow R$. Section 2.2 extends the semantics in ways that violate these properties. We are primarily interested in evaluating computations of returner type.

A CBPV *program* is a closed computation $M : \mathbf{F}\,\mathbf{bool}$. The reasoning principle we give for call-by-value and call-by-name relates open terms in program contexts. A *program relation* consists of a preorder[1] $\preccurlyeq$ on closed computations of type $\mathbf{F}\,\mathbf{bool}$. For example, we could use

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V{:}\mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \ \Rightarrow\ (M' \Downarrow \mathbf{return}\, V)$$

We could also use, for example, the total relation for $\preccurlyeq$ (and in this case apply our reasoning principle for call-by-value and call-by-name even if we include e.g. mutable state as a side effect – but then of course the conclusion of our reasoning principle would be trivial). Given any program relation $\preccurlyeq$, we define a *contextual preorder* $M \preccurlyeq^{\Gamma}_{\text{ctx}} M'$ on arbitrary well-typed computations (in typing context $\Gamma$) by considering the behaviour of $M$ and $M'$ in programs

[1] We do not actually need to assume that $\preccurlyeq$ is reflexive or transitive at any point, but because of constraints we add later (such as existence of an adequate model), it is unlikely that there are any interesting examples in which $\preccurlyeq$ is not a preorder.

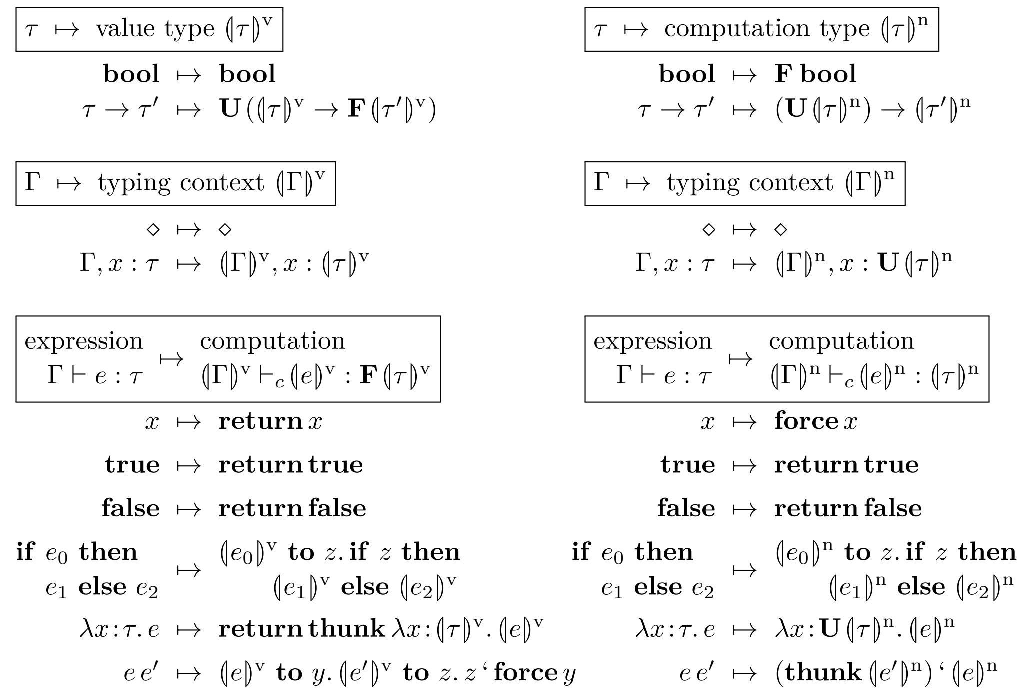


**Figure 3** Call-by-value (left) and call-by-name (right) translations into CBPV.

as follows. A *computation context* $\mathcal{E}$ is a computation term, with a single hole $\square$ where a computation term is expected. We write $\mathcal{E}[M]$ for the computation that results from replacing $\square$ with $M$ (which may capture some of the free variables of $M$).

▶ **Definition 1** (Contextual preorder). *Suppose that $\preccurlyeq$ is a program relation, and that $\Gamma \vdash_c M : \underline{C}$ and $\Gamma \vdash_c M' : \underline{C}$ are two computations of the same type. We write $M \preccurlyeq^{\Gamma}_{\text{ctx}} M'$ if, for all computation contexts $\mathcal{E}$ such that $\mathcal{E}[M], \mathcal{E}[M'] : \mathbf{F}\,\mathbf{bool}$, we have $\mathcal{E}[M] \preccurlyeq \mathcal{E}[M']$. We write $M \cong^{\Gamma}_{\text{ctx}} M'$, and say that $M$ and $M'$ are* contextually equivalent*, when both $M \preccurlyeq^{\Gamma}_{\text{ctx}} M'$ and $M' \preccurlyeq^{\Gamma}_{\text{ctx}} M$ hold.*

We sometimes omit $\Gamma$, and write just $M \preccurlyeq_{\text{ctx}} M'$ or $M \cong_{\text{ctx}} M'$.

## 2.1 Call-by-value and call-by-name

We use CBPV (instead of e.g. the monadic metalanguage [20]) because it captures both call-by-value and call-by-name evaluation. Levy [11] gives two compositional translations from a source language into CBPV: one for call-by-value and one for call-by-name. We recall both translations in this section; our goal is to reason about the relationship between them.

For the source language, we use the following syntax of types $\tau$ and expressions $e$:

$$\tau \;::=\; \mathbf{bool} \mid \tau \to \tau' \qquad e \;::=\; x \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \lambda x{:}\tau.\,e \mid e\,e'$$

The source language has a typing judgement of the form $\Gamma \vdash e : \tau$, defined by the usual rules.

The two translations from the source language to CBPV are defined in Figure 3. For call-by-value, each source language type $\tau$ is mapped to a CBPV value type $(\!|\tau|\!)^{\mathrm{v}}$ that contains the results of call-by-value computations. For call-by-name, $\tau$ is translated to a computation

type $(\!|\tau|\!)^{\mathrm{n}}$, which contains the computations themselves. Functions under the call-by-value translation accept values of type $(\!|\tau|\!)^{\mathrm{v}}$ as arguments; arguments are evaluated before being passed to the function. Under the call-by-name translation, functions accept thunks of computations as arguments; instead of evaluating them, arguments are thunked before passing them to call-by-name functions. Source-language typing contexts $\Gamma$ are translated to CBPV typing contexts $(\!|\Gamma|\!)^{\mathrm{v}}$ and $(\!|\Gamma|\!)^{\mathrm{n}}$. In call-by-value they contain values, in call-by-name they contain thunks of computations. Source-language expressions $e$ are mapped to CBPV computations $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$. The translation uses some auxiliary program variables, which are assumed fresh. Levy [11] proves that, in a precise sense, these translations do indeed capture call-by-value and call-by-name.

## 2.2 Examples

We consider three collections of (allowable) side-effects as examples throughout the paper.

► **Example 2** (No side-effects). We include the simplest possible example: the case where there are no side-effects at all. For this example, call-by-value and call-by-name turn out to have identical behaviour. We define the program relation $M \preccurlyeq M'$ (for closed computations $M, M' : \mathbf{F}\,\mathbf{bool}$) as:

$$M \preccurlyeq M' \quad \text{if and only if} \quad \exists V\!:\!\mathbf{bool}.\ (M \Downarrow \mathbf{return}\,V)\ \wedge\ (M' \Downarrow \mathbf{return}\,V)$$

In other words, $M$ and $M'$ both evaluate to the same result $V$. The contextual preorder $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ means if we construct two programs by wrapping $M$ and $M'$ in the same computation context, then these two programs evaluate to the same result. This relation is symmetric. Our other examples use non-symmetric relations.

► **Example 3** (Divergence). For our second example, the only side-effect is divergence (via recursion). In this case, call-by-value and call-by-name do not have identical behaviour (they are not related by $\preccurlyeq_{\mathrm{ctx}}$ as it is defined in our no-side-effects example). We instead show that replacing call-by-value with call-by-name does not change a terminating program into a diverging one.

We extend our two languages with recursion. For CBPV we extend the syntax of computations with fixed points $\mathbf{rec}\,x\!:\!\mathbf{U}\,\underline{C}.\,M$, and correspondingly extend the type system and operational semantics with the following rules:

$$\frac{\Gamma, x : \mathbf{U}\,\underline{C} \vdash_c M\ :\ \underline{C}}{\Gamma \vdash_c \mathbf{rec}\,x\!:\!\mathbf{U}\,\underline{C}.\,M\ :\ \underline{C}} \qquad \frac{M[x \mapsto \mathbf{thunk}\,(\mathbf{rec}\,x\!:\!\mathbf{U}\,\underline{C}.\,M)] \Downarrow R}{\mathbf{rec}\,x\!:\!\mathbf{U}\,\underline{C}.\,M \Downarrow R}$$

Of course, by adding recursion we lose normalization (but the semantics is still deterministic). We extend the source language, and the two translations into CBPV, with recursive functions:

$$e\ ::=\ \dots\ \mid \mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e \qquad \frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e : \tau \to \tau'}$$

$$\begin{aligned}(\!|\mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e|\!)^{\mathrm{v}} &= \mathbf{return}\,\mathbf{thunk}\,(\mathbf{rec}\,f\!:\!\mathbf{U}\,((\!|\tau|\!)^{\mathrm{v}} \to \mathbf{F}\,(\!|\tau'|\!)^{\mathrm{v}}).\,\lambda x\!:\!(\!|\tau|\!)^{\mathrm{v}}.\,(\!|e|\!)^{\mathrm{v}}) \\ (\!|\mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e|\!)^{\mathrm{n}} &= \mathbf{rec}\,f\!:\!\mathbf{U}\,(\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}} \to (\!|\tau'|\!)^{\mathrm{n}}).\,\lambda x\!:\!\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}.\,(\!|e|\!)^{\mathrm{n}}\end{aligned}$$

The expression $\Omega_\tau = ((\mathbf{rec}\,f\!:\!\mathbf{bool} \to \tau.\,\lambda x.\,f\,x)\,\mathbf{false}) : \tau$ enables us to distinguish between call-by-value and call-by-name: $(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau$ diverges in call-by-value but not in call-by-name. In particular, we have $(\!|(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,\mathbf{true}$, but there is no $R$ such that $(\!|(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau|\!)^{\mathrm{v}} \Downarrow R$.

For this example, we define the program relation $\preccurlyeq$ by

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V : \mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \ \Rightarrow\ (M' \Downarrow \mathbf{return}\, V)$$

so that $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ informally means if a program containing $M$ terminates with some result then the same program with $M'$ instead of $M$ terminates with the same result.

▶ **Example 4** (Nondeterminism)**.** Finally, we consider finite nondeterminism. Again call-by-value and call-by-name have different behaviour, but any result of a call-by-value execution is also a result of a call-by-name execution (if suitable nondeterministic choices are made).

We consider CBPV without recursion, but augmented with computations $\mathbf{fail}_{\underline{C}}$ for nullary nondeterministic choice and $M \,\mathbf{or}\, N$ for binary nondeterministic choice between computations; the typing and evaluation rules are standard:

$$\frac{}{\Gamma \vdash_c \mathbf{fail}_{\underline{C}} : \underline{C}} \qquad \frac{\Gamma \vdash_c M : \underline{C} \qquad \Gamma \vdash_c N : \underline{C}}{\Gamma \vdash_c M \,\mathbf{or}\, N : \underline{C}} \qquad \frac{M \Downarrow R}{M \,\mathbf{or}\, N \Downarrow R} \qquad \frac{N \Downarrow R}{M \,\mathbf{or}\, N \Downarrow R}$$

(There is no $R$ such that $\mathbf{fail}_{\underline{C}} \Downarrow R$.) We similarly include nullary and binary nondeterminism in the source language, and extend the call-by-value and call-by-name translations:

$$\frac{}{\Gamma \vdash \mathbf{fail}_\tau : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e \,\mathbf{or}\, e' : \tau} \qquad \begin{array}{l} (\!|\mathbf{fail}_\tau|\!)^{\mathrm{v}} = \mathbf{fail}_{\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}} \\ (\!|e \,\mathbf{or}\, e'|\!)^{\mathrm{v}} = (\!|e|\!)^{\mathrm{v}} \,\mathbf{or}\, (\!|e'|\!)^{\mathrm{v}} \end{array} \qquad \begin{array}{l} (\!|\mathbf{fail}_\tau|\!)^{\mathrm{n}} = \mathbf{fail}_{(\!|\tau|\!)^{\mathrm{n}}} \\ (\!|e \,\mathbf{or}\, e'|\!)^{\mathrm{n}} = (\!|e|\!)^{\mathrm{n}} \,\mathbf{or}\, (\!|e'|\!)^{\mathrm{n}} \end{array}$$

As an example, evaluating the expression $e = (\lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{true})(\mathbf{true}\ \mathbf{or}\ \mathbf{false})$ under call-by-value necessarily results in $\mathbf{true}$, but under call-by-name we can also get $\mathbf{false}$. (We have $(\!|e|\!)^{\mathrm{v}} \not\Downarrow \mathbf{return}\,\mathbf{false}$ but $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,\mathbf{false}$.)

For nondeterminism, we define $\preccurlyeq$ in the same way as our divergence example:

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V : \mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \ \Rightarrow\ (M' \Downarrow \mathbf{return}\, V)$$

This captures the property that any result that arises from an execution of $M$ (which may involve call-by-value) might arise from an execution of $M'$ (which may involve call-by-name).

## 3 Order-enriched denotational semantics

We give a denotational semantics for CBPV, which we use to prove instances of $\preccurlyeq_{\mathrm{ctx}}$. Since $\preccurlyeq_{\mathrm{ctx}}$ is not in general symmetric, we use *order-enriched* models, which come with partial orders $\sqsubseteq$ between denotations. In an *adequate* model, $[\![M]\!] \sqsubseteq [\![N]\!]$ implies $M \preccurlyeq_{\mathrm{ctx}} N$. Our semantics is based on Levy's *algebra models* [13] for CBPV, in which each computation type is interpreted as a monad algebra. (We restrict to algebra models for simplicity. Other forms of model, such as *adjunction models* [12] can be used for the same purpose.)

We assume no knowledge of enriched category theory; instead we give the relevant order-enriched (specifically **Poset**-enriched) definitions here. (We do however assume some basic ordinary category theory.)

▶ **Definition 5.** *A* **Poset**-category $\mathbf{C}$ *is an ordinary category, together with a partial order* $\sqsubseteq$ *on each hom-set* $\mathbf{C}(X, Y)$*, such that composition is monotone.*

If $\mathbf{C}$ is a **Poset**-category, we refer to the ordinary category as the *underlying* ordinary category, and write $|\mathbf{C}|$ for the class of objects.

▶ **Example 6.** We use the following three **Poset**-categories.

| **Poset**-category $\mathbf{C}$ | Objects $X \in |\mathbf{C}|$ | Morphisms $f : X \to Y$ | Order $f \sqsubseteq f'$ |
|---|---|---|---|
| **Set** | sets | functions | equality |
| **Poset** | posets | monotone functions | pointwise |
| $\omega$**Cpo** | $\omega$cpos | $\omega$-continuous functions | pointwise |

In each case, composition and identities are defined in the usual way. For **Set**, since the hom-posets $\mathbf{Set}(X, Y)$ are discrete, all of the **Poset**-enriched definitions coincide with the ordinary (unenriched) definitions. The objects of $\omega\mathbf{Cpo}$ are posets $(X, \sqsubseteq)$ for which $\sqsubseteq$ is $\omega$-*complete*, i.e. for which every $\omega$-chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ has a least upper bound $\bigsqcup x$. Morphisms are $\omega$-*continuous* functions, i.e. monotone functions that preserve least upper bounds of $\omega$-chains.

Let $\mathbf{C}$ be a **Poset**-category. We say that $\mathbf{C}$ is *cartesian* when its underlying category has a terminal object 1 and binary products $X_1 \times X_2$, such that the pairing functions $\langle -, - \rangle : \mathbf{C}(W, X_1) \times \mathbf{C}(W, X_2) \to \mathbf{C}(W, X_1 \times X_2)$ are monotone. When this is the case, there are canonical isomorphisms $assoc_{X_1,X_2,X_3} : (X_1 \times X_2) \times X_3 \to X_1 \times (X_2 \times X_3)$. We say that $\mathbf{C}$ is *cartesian closed* when it is cartesian and its underlying category has exponentials $X \Rightarrow Y$ for which the currying functions $\Lambda : \mathbf{C}(W \times X, Y) \to \mathbf{C}(W, X \Rightarrow Y)$ are monotone. (It follows that the uncurrying functions $\Lambda^{-1} : \mathbf{C}(W, X \Rightarrow Y) \to \mathbf{C}(W \times X, Y)$ are also monotone.) We write $ev_{X,Y}$ for the canonical morphism $\Lambda^{-1} id : (X \Rightarrow Y) \times X \to Y$. Binary *coproducts* in $\mathbf{C}$ are just binary coproducts in the underlying ordinary category, except that the copairing functions $[-, -] : \mathbf{C}(X_1, W) \times \mathbf{C}(X_2, W) \to \mathbf{C}(X_1 + X_2, W)$ are required to be monotone. The **Poset**-categories **Set**, **Poset**, and $\omega$**Cpo** are all cartesian closed, and have binary coproducts.

We interpret computation types as (Eilenberg-Moore) algebras for an order-enriched monad $\mathsf{T}$, which we need to be *strong* (just as models of Moggi's monadic metalanguage [20] use a strong monad). The definitions of strong **Poset**-monad and of $\mathsf{T}$-algebra we give are slightly non-standard, but are equivalent to the standard ones (see for example [17]). (In particular, it is more convenient for us to bake the strength into the Kleisli extension of the monad instead of having a separate strength.)

▶ **Definition 7** (Strong **Poset**-monad)**.** *A strong* **Poset**-*monad* $\mathsf{T} = (T, \eta, (-)^\dagger)$ *on a cartesian* **Poset**-*category* $\mathbf{C}$ *consists of an object* $TX \in |\mathbf{C}|$ *and morphism* $\eta_X : X \to TX$ *for each* $X \in |\mathbf{C}|$*, and a monotone function (*Kleisli extension*)* $(-)^\dagger : \mathbf{C}(W \times X, TY) \to \mathbf{C}(W \times TX, TY)$ *for each* $W, X, Y \in |\mathbf{C}|$*, such that*

$$\begin{aligned} f^\dagger \circ (id_W \times \eta_X) &= f : W \times X \to TY && \textit{for all } f : W \times X \to TY \\ (\eta_X \circ \pi_2)^\dagger &= \pi_2 : 1 \times TX \to TX && \textit{for all } X \in |\mathbf{C}| \\ (g^\dagger \circ (id_{W'} \times f) \circ assoc)^\dagger &= g^\dagger \circ (id_{W'} \times f^\dagger) \circ assoc && \textit{for all } f : W \times X \to TY, \\ &\quad : (W' \times W) \times TX \to TZ && \quad g : W' \times Y \to TZ \end{aligned}$$

Specializing the Kleisli extension of $\mathsf{T}$ to $W = 1$ produces a (non-strong) extension operator $(-)^\dagger : \mathbf{C}(X, TY) \to \mathbf{C}(TX, TY)$. We use this to define, for every $f : X \to Y$, a morphism $Tf : TX \to TY$ by $Tf = (\eta_Y \circ f)^\dagger$. (The latter definition makes $T$ into a **Poset**-*functor*.)

▶ **Definition 8** (Eilenberg-Moore algebra). *Let* $\mathsf{T}$ *be a strong* **Poset***-monad on a cartesian* **Poset***-category* $\mathbf{C}$*. A* $\mathsf{T}$-algebra $\mathsf{Z} = (Z, (-)^{\ddagger})$ *is a pair of an object* $Z \in |\mathbf{C}|$ *(the* carrier*) and monotone function (*extension *operator)* $(-)^{\ddagger} : \mathbf{C}(W \times X, Z) \to \mathbf{C}(W \times TX, Z)$ *for each* $W, X \in |\mathbf{C}|$*, such that*

$$f^{\ddagger} \circ (id_W \times \eta_X) = f : W \times X \to Z \qquad \text{for all } f : W \times X \to TY$$

$$(g^{\ddagger} \circ (id_{W'} \times f) \circ assoc)^{\ddagger} = g^{\ddagger} \circ (id_{W'} \times f^{\dagger}) \circ assoc \qquad \text{for all } f : W \times X \to TY,$$

$$: (W' \times W) \times TX \to Z \qquad g : W' \times Y \to Z$$

*For each* $X \in |\mathbf{C}|$*, we write* $F_{\mathsf{T}}X$ *for the* free $\mathsf{T}$-algebra $(TX, (-)^{\dagger})$*, and for each* $\mathsf{T}$*-algebra* $\mathsf{Z}$*, we write* $U_{\mathsf{T}}\mathsf{Z}$ *for the carrier* $Z \in |\mathbf{C}|$.

Specializing the extension operator of a $\mathsf{T}$-algebra $\mathsf{Z}$ to $W = 1$ produces a (non-strong) extension operator $(-)^{\ddagger} : \mathbf{C}(X, Z) \to \mathbf{C}(TX, Z)$.

Let $\mathsf{T}$ be a strong **Poset**-monad on a cartesian closed **Poset**-category $\mathbf{C}$. If $Y \in |\mathbf{C}|$ and $\mathsf{Z}$ is a $\mathsf{T}$-algebra, then there is a $\mathsf{T}$-algebra $Y \Rightarrow \mathsf{Z}$ with carrier $Y \Rightarrow Z$ and extension operator

$$f^{\ddagger} = \Lambda((\Lambda^{-1} f \circ \beta_{W,Y,X})^{\ddagger} \circ \beta_{W,TX,Y}) : W \times TX \to Y \Rightarrow Z \qquad \text{for } f : W \times X \to Y \Rightarrow Z$$

where $\beta_{X_1,X_2,X_3} = \langle\langle \pi_1 \circ \pi_1, \pi_2\rangle, \pi_2 \circ \pi_1\rangle : (X_1 \times X_2) \times X_3 \to (X_1 \times X_3) \times X_2$. These provide the interpretations of function types $A \to \underline{C}$.

▶ **Definition 9.** *A* model *of CBPV consists of*

- *a cartesian closed* **Poset***-category* $\mathbf{C}$*;*
- *the coproduct* $2 = 1 + 1$*, together with the corresponding morphisms* $inl, inr : 1 \to 2$*;*
- *a strong* **Poset***-monad* $\mathsf{T} = (T, \eta, (-)^{\dagger})$ *on* $\mathbf{C}$*.*

Given any model, the interpretation $[\![-]\!]$ of CBPV is defined in Figure 4. Value types $A$ are interpreted as objects $[\![A]\!] \in |\mathbf{C}|$, while computation types $\underline{C}$ are interpreted as $\mathsf{T}$-algebras. Typing contexts $\Gamma$ are interpreted as objects $[\![\Gamma]\!] \in \mathbf{C}$ using the cartesian structure of $\mathbf{C}$; if $(x : A) \in \Gamma$ then we write $\pi_x$ for the corresponding projection $[\![\Gamma]\!] \to [\![A]\!]$. Values $\Gamma \vdash V : A$ (respectively computations $\Gamma \vdash_c M : \underline{C}$) are interpreted as morphisms $[\![\Gamma \vdash V : A]\!]$ (resp. $[\![\Gamma \vdash_c M : \underline{C}]\!]$) in $\mathbf{C}$; we often omit the typing context and type when writing these. Programs $\diamond \vdash_c M : \mathbf{bool}$ are therefore interpreted as morphisms $[\![M]\!] : 1 \to T2$. To interpret **if**, we use the fact that, since $\mathbf{C}$ is cartesian closed, products distribute over the coproduct $2 = 1 + 1$. This means that for every $W \in |\mathbf{C}|$, the coproduct $W + W$ also exists in $\mathbf{C}$, and the canonical morphism

$$W + W \xrightarrow{[\langle id_W, inl \circ \langle\rangle_W\rangle, \langle id_W, inr \circ \langle\rangle_W\rangle]} W \times 2$$

has an inverse $dist_W : W \times 2 \to W + W$.

By composing the semantics of CBPV with the two translations of the source language, we obtain a call-by-value semantics $[\![-]\!]^{\mathrm{v}} = [\![(\!|-|\!)^{\mathrm{v}}]\!]$ and a call-by-name semantics $[\![-]\!]^{\mathrm{n}} = [\![(\!|-|\!)^{\mathrm{n}}]\!]$ of the source language.

We use the denotational semantics as a tool for proving instances of contextual preorders; for this we need *adequacy*.

▶ **Definition 10.** *A model of CBPV is* adequate *(with respect to a given program relation* $\preccurlyeq$*) if for all computations* $\Gamma \vdash_c M : \underline{C}$ *and* $\Gamma \vdash_c M' : \underline{C}$ *we have*

$$[\![\Gamma \vdash_c M : \underline{C}]\!] \sqsubseteq [\![\Gamma \vdash_c M' : \underline{C}]\!] \quad \Rightarrow \quad M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$$

**C**-object $[\![A]\!]$

$$[\![\mathbf{bool}]\!] = 2 \quad (= 1{+}1)$$
$$[\![\mathbf{U}\,\underline{C}]\!] = U_{\mathsf{T}}[\![\underline{C}]\!]$$

T-algebra $[\![\underline{C}]\!]$

$$[\![A \to \underline{C}]\!] = [\![A]\!] \Rightarrow [\![\underline{C}]\!]$$
$$[\![\mathbf{F}\,A]\!] = F_{\mathsf{T}}[\![A]\!]$$

**C**-object $[\![\Gamma]\!]$

$$[\![\diamond]\!] = 1$$
$$[\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

$[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$

$$[\![x]\!] = \pi_x$$
$$[\![\mathbf{true}]\!] = \mathit{inl} \circ \langle\rangle_{[\![\Gamma]\!]}$$
$$[\![\mathbf{false}]\!] = \mathit{inr} \circ \langle\rangle_{[\![\Gamma]\!]}$$
$$[\![\mathbf{thunk}\,M]\!] = [\![M]\!]$$

$[\![\Gamma \vdash_c M : \underline{C}]\!] : [\![\Gamma]\!] \to U_{\mathsf{T}}[\![\underline{C}]\!]$

$$[\![\lambda x{:}A.\,M]\!] = \Lambda[\![M]\!]$$
$$[\![V`M]\!] = \Lambda^{-1}[\![M]\!] \circ \langle \mathit{id}, [\![V]\!]\rangle$$
$$[\![\mathbf{return}\,V]\!] = \eta \circ [\![V]\!]$$
$$[\![M \ \mathbf{to}\ x.\,N]\!] = [\![N]\!]^{\ddagger} \circ \langle \mathit{id}, [\![M]\!]\rangle$$
$$[\![\mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2]\!] = [[\![M_1]\!], [\![M_2]\!]] \circ \mathit{dist} \circ \langle \mathit{id}, [\![V]\!]\rangle$$
$$[\![\mathbf{force}\,V]\!] = [\![V]\!]$$

**Figure 4** Denotational semantics of CBPV.

We give three different models, one for each of our three examples in Section 2.2. Each model is adequate with respect to the corresponding definition of $\preccurlyeq$; the proof in each case is a standard *logical relations* argument (e.g. [31]).

▶ **Example 11.** For CBPV with no side-effects, we use $\mathbf{C} = \mathbf{Set}$. The strong **Poset**-monad $\mathsf{T}$ is the identity on **Set**. Each $\mathsf{T}$-algebra $\mathsf{Z}$ is completely determined by its carrier $Z$; the extension operator $(-)^{\ddagger} : \mathbf{Set}(W \times X, Z) \to \mathbf{Set}(W \times X, Z)$ is necessarily the identity. The interpretation $[\![M]\!]$ of each program $M$ is just an element of 2.

▶ **Example 12.** For divergence, we use $\mathbf{C} = \omega\mathbf{Cpo}$. The strong **Poset**-monad $\mathsf{T}$ freely adjoins a least element $\bot$ to each $\omega\mathbf{Cpo}$. The unit $\eta_X$ is the inclusion $X \hookrightarrow TX$, while Kleisli extension is given by

$$f^{\dagger}(w,x) = \begin{cases} \bot & \text{if } x = \bot \\ f(w,x) & \text{otherwise} \end{cases}$$

A $\mathsf{T}$-algebra $\mathsf{Z}$ is equivalently an $\omega\mathbf{Cpo}$ $Z$ with a least element $\bot \in Z$. The extension operator is completely determined once the carrier is fixed; it is analogous to $(-)^{\dagger}$. Since the exponential $Y \Rightarrow Z$ is the set of $\omega$-continuous functions $Y \to Z$ ordered pointwise, it has a least element (forms a $\mathsf{T}$-algebra) whenever $Z$ does.

If $\mathsf{Z}$ is a $\mathsf{T}$-algebra, then every $\omega$-continuous function $f : Z \to Z$ has a least fixed point $\mathit{fix}\,f = \bigsqcup_{n\in\mathbb{N}} f^n\bot \in Z$. These enable us to interpret recursive computations, by defining $[\![\mathbf{rec}\,x{:}\mathbf{U}\underline{C}.\,M]\!]\rho = \mathit{fix}\big(x \mapsto [\![M]\!](\rho, x)\big)$. The interpretation $[\![M]\!]$ of a program $M : \mathbf{F}\,\mathbf{bool}$ is either $\bot$ (signifying divergence), or one of the two elements of 2.

▶ **Example 13.** For finite nondeterminism, we use $\mathbf{C} = \mathbf{Poset}$. The strong **Poset**-monad $\mathsf{T}$ freely adds finite joins to each poset. It is defined by

$$TX = (\{\downarrow S' \mid S' \in \mathcal{P}_{\text{fin}} X\}, \subseteq) \qquad \eta_X\, x = \downarrow\{x\} \qquad f^{\dagger}(w,S) = \textstyle\bigcup_{x\in S} f(w,x)$$

where $\mathcal{P}_{\text{fin}} X$ is the set of finite subsets of $X$, and $\downarrow S' = \{x \in X \mid \exists x' \in S'.\, x \sqsubseteq x'\}$ is the *downwards-closure* of $S' \subseteq X$. Each $\mathsf{T}$-algebra is again completely determined by its carrier; a $\mathsf{T}$-algebra $\mathsf{Z}$ is equivalently a poset $Z$ that has finite joins. The extension operator is necessarily given by $f^{\ddagger}(w, S) = \bigsqcup_{x \in S} f(w, x)$. (The latter join exists because $S$ is the downwards-closure of a finite set, even though $S$ itself might not be finite.) The function space $Y \Rightarrow Z$ is the set of monotone functions $Y \to Z$, ordered pointwise; if $Z$ has finite joins then $Y \Rightarrow Z$ has finite joins computed pointwise.

We interpret nondeterministic computations using nullary and binary joins:

$$[\![\mathbf{fail}_{\underline{C}}]\!]\rho = \bot \qquad [\![M \ \mathbf{or}\ N]\!]\rho = [\![M]\!]\rho \sqcup [\![N]\!]\rho$$

The interpretation $[\![M]\!]$ of a program $M : \mathbf{F}\,\mathbf{bool}$ is one of the four subsets of 2.

## 4 A Galois connection between call-by-value and call-by-name

We now return to the main contribution of this paper: relating call-by-value and call-by-name. The main difficulty in doing this is that the call-by-value and call-by-name translations of expressions have different types:

$$(\!|\Gamma|\!)^{\mathrm{v}} \vdash_c (\!|e|\!)^{\mathrm{v}} : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \qquad (\!|\Gamma|\!)^{\mathrm{n}} \vdash_c (\!|e|\!)^{\mathrm{n}} : (\!|\tau|\!)^{\mathrm{n}}$$

We cannot ask whether $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ are related by $\preccurlyeq_{\mathrm{ctx}}$, because $\preccurlyeq_{\mathrm{ctx}}$ only relates terms of the same type. It does not make sense to replace $(\!|e|\!)^{\mathrm{v}}$ with $(\!|e|\!)^{\mathrm{n}}$ inside a CBPV program, because the result would not be well-typed.

Reynolds [25] solves a similar problem when comparing direct and continuation semantics of the $\lambda$-calculus by defining maps between the two semantics, so that a denotation in the direct semantics can be viewed as a denotation in the continuation semantics and vice versa. We use the same idea here. Specifically, we define maps $\Phi$ from call-by-value computations to call-by-name computations, and $\Psi$ from call-by-name to call-by-value:[2]

$$\Gamma \vdash_c M : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \ \mapsto\ \Gamma \vdash_c \Phi_\tau M : (\!|\tau|\!)^{\mathrm{n}} \qquad\qquad \Gamma \vdash_c N : (\!|\tau|\!)^{\mathrm{n}} \ \mapsto\ \Gamma \vdash_c \Psi_\tau N : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

Then instead of replacing $(\!|e|\!)^{\mathrm{v}}$ with $(\!|e|\!)^{\mathrm{n}}$ directly, we use $\Phi$ and $\Psi$ to convert $(\!|e|\!)^{\mathrm{n}}$ to a computation of the same type as $(\!|e|\!)^{\mathrm{v}}$ (we define this computation formally in Section 5):

$$(\!|\Gamma|\!)^{\mathrm{v}} \longrightarrow (\!|\Gamma|\!)^{\mathrm{n}} \xrightarrow{(\!|e|\!)^{\mathrm{n}}} (\!|\tau|\!)^{\mathrm{n}} \longrightarrow \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

This behaves like a call-by-name computation, but has the same type as a call-by-value computation. We do not want just *any* maps between call-by-value and call-by-name. We show that, under certain conditions (the assumptions of Theorem 17 below) the maps we define form *Galois connections* [18] in the denotational semantics. This is crucial for the correctness of our reasoning principle. The conditions needed to show that we have Galois connections are where the choice of side-effects becomes important.

---

[2] We define $\Phi$ and $\Psi$ directly as maps from computations to computations, but we could instead have defined computations

$$x : \mathbf{U}\,\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \vdash_c \Phi'_\tau : (\!|\tau|\!)^{\mathrm{n}} \qquad x : \mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}} \vdash_c \Psi'_\tau : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

and then recovered $\Phi$ and $\Psi$ using substitution (up to the equational theory defined in Appendix A). This definition is slightly less convenient to work with however.

▶ **Definition 14.** *A* Galois connection *consists of two posets $X$, $Y$ and two monotone functions $\phi : X \to Y$, $\psi : Y \to X$, such that $x \sqsubseteq \psi(\phi\, x)$ for all $x \in X$ and $\phi(\psi\, y) \sqsubseteq y$ for all $y \in Y$.*

The syntactic maps $\Phi_\tau$ and $\Psi_\tau$ are defined as follows. (We use some extra variables in the definition, which are assumed to be fresh.)

$$\Phi_\tau M = M \text{ **to** } x.\, \hat{\Phi}_\tau x$$

$$\hat{\Phi}_{\mathbf{bool}} V = \mathbf{return}\, V \qquad \hat{\Phi}_{\tau\to\tau'} V = \lambda x{:}\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}.\, \Psi_\tau(\mathbf{force}\, x) \text{ **to** } y.\ (y\,`\,\mathbf{force}\, V) \text{ **to** } z.\, \hat{\Phi}_{\tau'} z$$

$$\Psi_{\mathbf{bool}} N = N \qquad \Psi_{\tau\to\tau'} N = \mathbf{return}\,\mathbf{thunk}\,\lambda x{:}(\!|\tau|\!)^{\mathrm{v}}.\, \Psi_{\tau'}\big((\mathbf{thunk}\,(\hat{\Phi}_\tau x))\,`\,N\big)$$

The maps $\Phi_\tau$ from call-by-value computations first evaluate the computation, and then map the result to call-by-name using $\hat{\Phi}_\tau$, which has the following typing:

$$\Gamma \vdash V : (\!|\tau|\!)^{\mathrm{v}} \quad \mapsto \quad \Gamma \vdash_c \hat{\Phi}_\tau V : (\!|\tau|\!)^{\mathrm{n}}$$

Given any model of CBPV, we correspondingly define morphisms $\phi_\tau : T[\![\tau]\!]^{\mathrm{v}} \to U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}$ and $\psi_\tau : U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}} \to T[\![\tau]\!]^{\mathrm{v}}$ as follows (where $\hat{\phi}_\tau : [\![\tau]\!]^{\mathrm{v}} \to U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}$).

$$\phi_\tau = \hat{\phi}_\tau^{\ddagger} \qquad \hat{\phi}_{\mathbf{bool}} = \eta_2 \qquad \hat{\phi}_{\tau\to\tau'} = \Lambda((\phi_{\tau'} \circ ev)^{\ddagger} \circ (id_{[\![\tau\to\tau']\!]^{\mathrm{v}}} \times \psi_\tau))$$

$$\psi_{\mathbf{bool}} = id_{T2} \qquad \psi_{\tau\to\tau'} = \eta_{[\![\tau\to\tau']\!]^{\mathrm{v}}} \circ (\hat{\phi}_\tau \Rightarrow \psi_{\tau'})$$

These morphisms are the interpretations of $\Phi$ and $\Psi$ in the following sense.

▶ **Lemma 15.** *Given any model of CBPV, if $\Gamma \vdash_c M : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$ then $[\![\Phi_\tau M]\!] = \phi_\tau \circ [\![M]\!]$, and if $\Gamma \vdash_c N : (\!|\tau|\!)^{\mathrm{n}}$ then $[\![\Psi_\tau N]\!] = \psi_\tau \circ [\![N]\!]$.*

**Proof sketch.** By induction on the type $\tau$. Each case is an easy calculation. ◀

For the rest of this section, we show that in every model of CBPV that satisfies certain conditions (the assumptions of Theorem 17 below), the functions

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \quad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

form a Galois connection for every $\tau$ and $W \in |\mathbf{C}|$. This is the key step in the proof of our reasoning principle (Theorem 21).

First we note that we cannot expect these maps to form Galois connections in general. Consider what happens when we convert a CBPV computation $M : \mathbf{F}\,(\!|\mathbf{bool} \to \mathbf{bool}|\!)^{\mathrm{v}} = \mathbf{F}\,(\mathbf{U}\,(\mathbf{bool} \to \mathbf{F}\,\mathbf{bool}))$ to call-by-name and then back to call-by-value. The computation $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M)$ that results is essentially[3] the same as

$$\mathbf{return}\,\mathbf{thunk}\,\lambda x{:}\mathbf{bool}.\, M \text{ **to** } z.\, x\,`\,\mathbf{force}\, z$$

The computation $M$ may cause side-effects before producing a (thunk of a) function; but $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M)$ does not. Thus in general (e.g. if side-effects include mutable state), we cannot expect to have $[\![M]\!] \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}} \circ \phi_{\mathbf{bool}\to\mathbf{bool}} \circ [\![M]\!]$, and hence cannot expect to have a Galois connection. The round-trip from call-by-value to call-by-name and back thunks the side-effects of $M$.

The inequality $[\![M]\!] \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}}(\phi_{\mathbf{bool}\to\mathbf{bool}}[\![M]\!])$ does however hold when $[\![M]\!]$ is *lax thunkable* in the following sense.

[3] Precisely, they are the same in the sense that $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M) \equiv \mathbf{return}\,\mathbf{thunk}\,\lambda x : \mathbf{bool}.\, M \text{ **to** } z.\, x\,`\,\mathbf{force}\, z$, where $\equiv$ is the equational theory defined in Appendix A.

▶ **Definition 16.** *Let* $\mathsf{T}$ *be a strong* **Poset***-monad on a cartesian* $\mathbf{C}$*. We say that a morphism* $f : X \to TY$ *is* lax thunkable *when* $T\eta_Y \circ f \sqsubseteq \eta_{TY} \circ f$*. We say that* $\mathsf{T}$ *is* lax idempotent[4] *when* $T\eta_Y \sqsubseteq \eta_{TY}$ *for all* $Y \in |\mathbf{C}|$ *(equivalently, when every such morphism is lax thunkable).*

Our notion of *lax thunkable* morphism is an inequational version of Führmann's [5] notion of *thunkable* morphism. We do not need it below, but we can give a corresponding definition for CBPV: a computation $\Gamma \vdash_c M : \mathbf{F}\,A$ is *lax thunkable* (with respect to a given $\preccurlyeq$) when

$$M\ \mathbf{to}\ x.\,\mathbf{return}\,\mathbf{thunk}\,\mathbf{return}\,x \ \preccurlyeq^{\Gamma}_{\mathrm{ctx}}\ \mathbf{return}\,\mathbf{thunk}\,M$$

This is the case in particular when the interpretation $[\![M]\!] : [\![\Gamma]\!] \to T[\![A]\!]$ of $M$, in an adequate model, is a lax thunkable morphism.

Assuming that $\mathsf{T}$ is lax idempotent is enough to achieve the goal of this section:

▶ **Theorem 17.** *Given a CBPV model in which* $\mathsf{T}$ *is lax idempotent, the functions*

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \quad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

*form a Galois connection for every source-language type* $\tau$ *and object* $X \in |\mathbf{C}|$*.*

**Proof sketch.** By induction on the type $\tau$. This is trivial for **bool**. For function types both of the required inequalities use the fact that $\mathsf{T}$ is lax idempotent. ◀

▶ **Example 18.** Each of our three example models is adequate, and has a lax idempotent $\mathsf{T}$. For no side-effects, we use the identity monad, which is trivially lax idempotent because $T\eta_Y = id_Y = \eta_{TY}$. For divergence, the monad is lax idempotent because the left hand side of $T\eta_Y t \sqsubseteq \eta_{TY} t$ is $\bot$ when $t = \bot$ (intuitively, we can thunk diverging computations), and otherwise the two sides are equal. For nondeterminism, we have

$$T\eta_Y S = {\downarrow}\{{\downarrow}\{y\} \mid y \in S\} \subseteq {\downarrow}\{S\} = \eta_{TY} S$$

because ${\downarrow}\{y\} \subseteq S$ for every $y \in S$ (intuitively, we can postpone nondeterministic choices). Thus we obtain Galois connections in each of these three cases.

▶ Remark 19. Given an adequate model in which $\mathsf{T}$ is lax idempotent, it follows from Theorem 17 that the maps $\Phi_\tau$ and $\Psi_\tau$ on terms form a Galois connection (with respect to $\preccurlyeq_{\mathrm{ctx}}$), by Lemma 15. In particular, we have

$$M \preccurlyeq_{\mathrm{ctx}} \Psi_\tau(\Phi_\tau M) \qquad \Phi_\tau(\Psi_\tau N) \preccurlyeq_{\mathrm{ctx}} N$$

Both of these inequalities are in general proper (they are not contextual equivalences). To see this, consider our divergence example, for which the above inequalities hold. For each $\underline{C}$, let $\Omega_{\underline{C}}$ be the diverging computation $\mathbf{rec}\,x : \mathbf{U}\,\underline{C}.\,\mathbf{force}\,x$ (which has type $\underline{C}$). Then if $\tau = \mathbf{bool} \to \mathbf{bool}$ and $M = \Omega_{\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}}$, we do not have $M \succcurlyeq_{\mathrm{ctx}} \Psi_\tau(\Phi_\tau M)$, because for $\mathcal{E} = (\Box\ \mathbf{to}\ f.\,\mathbf{return}\,\mathbf{false})$ the computation $\mathcal{E}[M]$ diverges but $\mathcal{E}[\Psi_\tau(\Phi_\tau M)] \Downarrow \mathbf{return}\,\mathbf{false}$. In this case we have $\Psi_\tau(\Phi_\tau M) \cong_{\mathrm{ctx}} \mathbf{return}\,\mathbf{thunk}\,\lambda x : \mathbf{bool}.\,\Omega_{\mathbf{F}\,\mathbf{bool}}$. For a counterexample to $\Phi_\tau(\Psi_\tau N) \succcurlyeq_{\mathrm{ctx}} N$, let $\tau = \mathbf{bool} \to \mathbf{bool}$ and $N = \lambda x : \mathbf{U}\,\mathbf{F}\,\mathbf{bool}.\,\mathbf{return}\,\mathbf{true}$. Then for $\mathcal{E}' = ((\mathbf{thunk}\,\Omega_{\mathbf{F}\,\mathbf{bool}})\,\text{`}\,\Box)$, the computation $\mathcal{E}'[\Phi_\tau(\Psi_\tau N)]$ diverges but $\mathcal{E}'[N] \Downarrow \mathbf{return}\,\mathbf{true}$. Here we have $\Phi_\tau(\Psi_\tau N) \cong_{\mathrm{ctx}} \lambda x : \mathbf{U}\,\mathbf{F}\,\mathbf{bool}.\,\mathbf{force}\,x\ \mathbf{to}\ y.\,\mathbf{return}\,\mathbf{true}$.

[4] Lax idempotent **Poset**-monads are a special case of lax idempotent 2-monads, which are well-known, and are often called *Kock-Zöberlein* monads [8].

## 5 The reasoning principle

We now use the Galois connections defined in the previous section to relate the call-by-value and call-by-name translations of expressions, and arrive at our main reasoning principle.

Recall that we compose the call-by-name translation of each $e$ with the maps $\Phi$ and $\Psi$ defined above, to form a CBPV computation of the same type as the call-by-value translation:

$$(\!|\Gamma|\!)^{\mathrm{v}} \longrightarrow (\!|\Gamma|\!)^{\mathrm{n}} \xrightarrow{(\!|e|\!)^{\mathrm{n}}} (\!|\tau|\!)^{\mathrm{n}} \longrightarrow \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

We first define this composition precisely. The arrow on the right is just given by applying $\Psi_\tau$. The arrow on the left is a substitution that maps terms in call-by-name contexts $(\!|\Gamma|\!)^{\mathrm{n}}$ to terms in call-by-value contexts $(\!|\Gamma|\!)^{\mathrm{v}}$. Given any source-language typing context $\Gamma = x_1 : \tau_1, \dots x_n : \tau_n$, we define $\hat{\Phi}_\Gamma \;=\; x_1 \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau_1} x_1), \dots, x_n \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau_n} x_n)$. If $\Gamma \vdash e : \tau$ then $(\!|\Gamma|\!)^{\mathrm{v}} \vdash_c \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big) : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$, which has the same typing as $(\!|e|\!)^{\mathrm{v}}$. The statement we wish to prove is $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$. Again we reason using the denotational semantics. Given a CBPV model, the interpretation of the substitution $\hat{\Phi}_\Gamma$ is a morphism $\hat{\phi}_\Gamma : [\![\Gamma]\!]^{\mathrm{v}} \to [\![\Gamma]\!]^{\mathrm{n}}$, given by $\hat{\phi}_\diamond = id_1$ and $\hat{\phi}_{\Gamma,x:\tau} = \hat{\phi}_\Gamma \times \hat{\phi}_\tau$. If the model is adequate, to show our goal $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$, it suffices to show that $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$.

We show that this is the case directly using the properties of Galois connections, which allow us to push composition with $\psi_\tau$ into the structure of terms.

▶ **Lemma 20.** *In every CBPV model for which the functions*

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \quad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

*form Galois connections for all $\tau$, $W$, we have $[\![e]\!]^{\mathrm{v}} \;\sqsubseteq\; \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ for all $\Gamma \vdash e : \tau$.*

**Proof sketch.** By induction on the derivation of $\Gamma \vdash e : \tau$. We give just the case for function applications $e\,e'$; which shows where having Galois connections is useful. The two inequalities below both use properties of Galois connections. The equalities follow from properties of products, exponentials, and $\mathsf{T}$-algebras.

$$\begin{aligned}
[\![e\,e']\!]^{\mathrm{v}} &= \big(ev^\dagger \circ \langle \pi_2, [\![e']\!]^{\mathrm{v}} \circ \pi_1\rangle\big)^\dagger \circ \langle id, [\![e]\!]^{\mathrm{v}}\rangle \\
&\sqsubseteq \psi_{\tau'} \circ \phi_{\tau'} \circ \big(ev^\dagger \circ \langle \pi_2, \psi_\tau \circ [\![e']\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma \circ \pi_1\rangle\big)^\dagger \circ \langle id, \psi_{\tau\to\tau'} \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma\rangle \\
&= \psi_{\tau'} \circ \Lambda^{-1}(\phi_{\tau\to\tau'} \circ \psi_{\tau\to\tau'} \circ [\![e]\!]^{\mathrm{n}}) \circ \langle id, [\![e']\!]^{\mathrm{n}}\rangle \circ \hat{\phi}_\Gamma \\
&\sqsubseteq \psi_{\tau'} \circ \Lambda^{-1}[\![e]\!]^{\mathrm{n}} \circ \langle id, [\![e']\!]^{\mathrm{n}}\rangle \circ \hat{\phi}_\Gamma \\
&= \psi_{\tau'} \circ [\![e\,e']\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma
\end{aligned}$$

◀

Theorem 17 provides a sufficient condition for the maps between the two evaluation orders to form Galois connections. By combining this sufficient condition with the above lemma, we arrive at our reasoning principle, which we state formally as Theorem 21. Recall that a program relation $\preccurlyeq$ is a family of relations on CBPV programs, and that each program relation induces a contextual preorder $\preccurlyeq_{\mathrm{ctx}}$. Given any program relation $\preccurlyeq$, to show that the call-by-value and call-by-name translations of source-language expressions are related by $\preccurlyeq_{\mathrm{ctx}}$ it is enough to find an adequate model involving a lax idempotent $\mathsf{T}$.

▶ **Theorem 21** (Relationship between call-by-value and call-by-name)**.** *Suppose we are given a program relation $\preccurlyeq$, and a model of CBPV that is adequate with respect to $\preccurlyeq$, and has a lax idempotent $\mathsf{T}$. If $\Gamma \vdash e : \tau$ then*

$$(\!|e|\!)^{\mathrm{v}} \;\preccurlyeq_{\mathrm{ctx}}\; \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$$

**Proof.** Theorem 17 implies that $\phi$ and $\psi$ form Galois connections, and then Lemma 20 implies $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$. The result follows from adequacy of the model and Lemma 15. ◀

The generality of this theorem comes from two sources. First, we consider arbitrary program relations $\preccurlyeq$. The only requirement on these is the existence of some adequate model in which morphisms are lax thunkable. Second, this theorem applies to terms that are open and have higher types, using the maps between the two evaluation orders. We obtain a corollary about source-language *programs* (closed expressions of type **bool**). This corollary is closer to the standard results that are proved for specific side-effects.

▶ **Corollary 22.** *If the assumptions of Theorem 21 hold, then for every closed expression $e$ of type* **bool**, *we have* $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$.

**Proof.** We have $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_{\mathbf{bool}} \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\diamond = [\![e]\!]^{\mathrm{n}}$ because both $\psi_{\mathbf{bool}}$ and $\hat{\phi}_\diamond$ are identities. Adequacy implies $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} (\!|e|\!)^{\mathrm{n}}$, and hence $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$. ◀

Our reasoning principle also has a partial converse:

▶ **Lemma 23.** *If* $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ *for each* $\Gamma \vdash e : \tau$, *then* $T\eta_2 \sqsubseteq \eta_{T2}$, *and every morphism* $X \to T2$ *is lax thunkable.*

**Proof sketch.** The first step is to show that $id \sqsubseteq \psi_\tau \circ \phi_\tau$ for every $\tau$, by applying the assumption to the expression $x : \mathbf{bool} \to \tau \vdash x\,\mathbf{false} : \tau$. (It does not matter whether we use **false** or use **true**; we could have used $x : \mathbf{unit} \to \tau$ if we had a unit type.) Then, since $\phi_{\mathbf{bool}}$ and $\psi_{\mathbf{bool}}$ are identities, we get $id_{T(2 \Rightarrow T2)} \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}} \circ \phi_{\mathbf{bool}\to\mathbf{bool}} = \eta_{2\Rightarrow T2} \circ id_{2\Rightarrow T2}{}^{\ddagger}$, from which the result follows. ◀

As a final remark, while we compose the call-by-value translation on both sides, this choice is in fact arbitrary. By properties of Galois connections, the inequality $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ is equivalent to $\phi_\tau \circ [\![e]\!]^{\mathrm{v}} \sqsubseteq [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$, and two other inequalities are available when $\mathsf{T}$ is lax idempotent by defining suitable morphisms $\psi_\Gamma : [\![\Gamma]\!]^{\mathrm{n}} \to T[\![\Gamma]\!]^{\mathrm{v}}$.

We now return to our three examples. For each example, we take the adequate model defined in Section 3; in all three cases, the strong **Poset**-monad $\mathsf{T}$ is lax idempotent. After extending the inductive proof of Lemma 20 with cases for the extra syntax, we can apply our relationship between call-by-value and call-by-name (Theorem 21).

In particular, we can apply Corollary 22 to relate source-language programs. For no side-effects, this shows for each $e : \mathbf{bool}$ that there is some $V$ such that $(\!|e|\!)^{\mathrm{v}} \Downarrow \mathbf{return}\,V$ and $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,V$. In other words, $e$ evaluates to the same result in call-by-value and in call-by-name (since evaluation is deterministic). For divergence and for nondeterminism, the corollary says that $(\!|e|\!)^{\mathrm{v}} \Downarrow \mathbf{return}\,V$ implies $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,V$ for all $V$. Hence for divergence, if the call-by-value execution terminates with some result, the call-by-name execution terminates with the same result. For nondeterminism, all possible results of call-by-value executions are possible results of call-by-name executions.

## 6 Related work

**Comparing evaluation orders.** Plotkin [24] and many others (e.g. [7]) relate call-by-value and call-by-name. Crucially, they consider lambda-calculi with no side-effects other than divergence. This makes a significant difference to the techniques that can be used, in particular because in this case the equational theory for call-by-name is strictly weaker than for call-by-value. This is not necessarily true for other side-effects. Other evaluation orders

(such as call-by-need) have also been compared in similarly restricted settings [14, 15, 6]. We suspect our technique could also be adapted to these. Here we use CBPV as a calculus in which to reason about both call-by-value and call-by-name, but other calculi (e.g. the modal calculus of [28]) may be suitable for this purpose.

It might also be possible to recast some of our work in terms of the *duality* between call-by-value and call-by-name [3, 2, 30, 29], In particular, this may shed some light on our definitions of $\Phi$ and $\Psi$. It is not clear to us what the precise connection is however. While Selinger [29] defines translations between call-by-value and call-by-name versions of Parigot's $\lambda\mu$-calculus [23], these translations behave differently to ours, in particular, they are semantics-preserving.

**Relating semantics of languages.** The technique we use here to relate call-by-value and call-by-name is based on the idea used first by Reynolds [25] to relate direct and continuation semantics of the lambda calculus, and later used by others (e.g. [19, 9, 1, 4]). There are several differences with our approach. Reynolds constructs a logical relation between the two semantics, and uses this to establish a relationship with the two maps. We skip the logical relation step. Reynolds also relies on continuations with a large-enough domain of answers (e.g. a solution to a particular recursive domain equation). Our maps exist for *any* choice of model. We are the first to use this technique to relate call-by-value and call-by-name. There has been some work [26, 10, 27] on soundness and completeness properties of translations (similar to the translations into CBPV), in particular using Galois connections (and similar structures) for which the order is reduction of programs. Our results would fail if we used reduction of programs directly, so we consider only the observable behaviour of programs.

There are some similarities between our work and the work of New et al. [21, 22] on gradual typing. In particular, [22] has embedding-projection pairs (a special case of Galois connections) for casting from a more dynamic type to a less dynamic type, and vice versa. Their application is quite different however. The double category perspective used in [21] may also be illuminating here.

## 7 Conclusions

In this paper, we give a general reasoning principle (Theorem 21) that relates the observable behaviour of terms under call-by-value and call-by-name. The reasoning principle works for various collections of side-effects, in particular, it enables us to obtain theorems about divergence and nondeterminism. It is about open expressions, and allows us to change evaluation order *within* programs. We obtain a result about call-by-value and call-by-name evaluations of *programs* as a corollary (Corollary 22). Applying this to divergence, we show that if the call-by-value execution terminates with some result then the call-by-name execution terminates with the same result. For nondeterminism, we show that all possible results of call-by-value executions are possible results of call-by-name executions. There may be other collections of side-effects we can apply our technique to, including combinations of divergence and nondeterminism.

We expect that our technique can be applied to other evaluation orders. Two evaluation orders can be related by giving translations into some common language (here we use CBPV), constructing maps between the two translations, and showing that (for some models) these maps form Galois connections. A major advantage of the technique is that it allows us to identify axiomatic properties of side-effects (thunkable, etc.) that give rise to relationships between evaluation orders.

## References


1. Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994. doi:10.1007/3-540-57887-0_99.
2. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 233–243. ACM, 2000. doi:10.1145/351240.351262.
3. Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, 1989.
4. Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.
5. Carsten Führmann. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 20:245–292, 1999. doi:10.1016/S1571-0661(04)80078-1.
6. Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP):114:1–114:23, 2019. doi:10.1145/3341718.
7. Jun Inoue and Walid Taha. Reasoning about multi-stage programs. *Journal of Functional Programming*, 26(e22), 2016. doi:10.1017/S0956796816000253.
8. Anders Kock. Monads for which structures are adjoint to units. *Journal of Pure and Applied Algebra*, 104(1):41–59, 1995. doi:10.1016/0022-4049(94)00111-U.
9. Jakov Kučan. Retraction approach to cps transform. *Higher Order Symbol. Comput.*, 11(2):145–175, 1998. doi:10.1023/A:1010012532463.
10. Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 124–136. ACM, 1993. doi:10.1145/158511.158613.
11. Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999. doi:10.1007/3-540-48959-2_17.
12. Paul Blain Levy. Adjunction models for call-by-push-value with stacks. *Electronic Notes in Theoretical Computer Science*, 69:248–271, 2003. CTCS'02, Category Theory and Computer Science. doi:10.1016/S1571-0661(04)80568-1.
13. Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006. doi:10.1007/s10990-006-0480-6.
14. John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *Proceedings of the Eleventh Annual Mathematical Foundations of Programming Semantics Conference*, pages 370–392, 1995. doi:10.1016/S1571-0661(04)00022-2.
15. Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In Luís Caires, editor, *Programming Languages and Systems*, pages 235–262. Springer, 2019. doi:10.1007/978-3-030-17184-1_9.
16. Dylan McDermott and Alan Mycroft. Galois connecting call-by-value and call-by-name (extended version), 2022. arXiv:2202.08246.
17. Dylan McDermott and Tarmo Uustalu. What makes a strong monad? In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming (to appear)*. Open Publishing Association, 2022.
18. Austin Melton, David A. Schmidt, and George E. Strecker. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. Springer, 1986. doi:10.1007/3-540-17162-2_130.
19. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In Rohit Parikh, editor, *Logics of Programs*, pages 219–224. Springer, 1985. doi:10.1007/3-540-15648-8_17.
20. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

21 Max S. New and Daniel R. Licata. Call-by-name gradual type theory. *Logical Methods in Computer Science*, 16, 2020. doi:10.23638/LMCS-16(1:7)2020.
22 Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *Journal of Functional Programming*, 31, 2021. doi:10.1017/S0956796821000125.
23 Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 190–201. Springer, 1992. doi:10.1007/BFb0013061.
24 G. 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.
25 John C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 141–156. Springer, 1974. doi:10.1007/978-3-662-21545-6_10.
26 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 288–298. ACM, 1992. doi:10.1145/141471.141563.
27 Amr Sabry and Philip Wadler. A reflection on call-by-value. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 13–24. ACM, 1996. doi:10.1145/232627.232631.
28 José Espírito Santo, Luís Pinto, and Tarmo Uustalu. Plotkin's call-by-value $\lambda$-calculus as a modal calculus. *Journal of Logical and Algebraic Methods in Programming*, 2022. doi:10.1016/j.jlamp.2022.100775.
29 Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001. doi:10.1017/S096012950000311X.
30 Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 189–201. ACM, 2003. doi:10.1145/944705.944723.
31 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. doi:10.7551/mitpress/3054.001.0001.

## A CBPV equational theory

$$\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \equiv M_1 \qquad V\text{`}\lambda x{:}A.\,M \equiv M[x \mapsto V]$$
$$\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \equiv M_2 \qquad \mathbf{force}\,\mathbf{thunk}\,M \equiv M$$

$$\begin{aligned} V &\equiv \mathbf{thunk}\,\mathbf{force}\,V \\ M[x \mapsto V] &\equiv \mathbf{if}\ V\ \mathbf{then}\ M[x \mapsto \mathbf{true}]\ \mathbf{else}\ M[x \mapsto \mathbf{false}] \\ M &\equiv \lambda x{:}A.\,x\text{`}M \end{aligned}$$

$$\mathbf{return}\,V\ \mathbf{to}\ x.\,M \equiv M[x \mapsto V]$$
$$M \equiv M\ \mathbf{to}\ x.\,\mathbf{return}\,x$$
$$(M_1\ \mathbf{to}\ x.\,M_2)\ \mathbf{to}\ y.\,M_3 \equiv M_1\ \mathbf{to}\ x.\,(M_2\ \mathbf{to}\ y.\,M_3)$$
$$\lambda y{:}A.\,M\ \mathbf{to}\ x.\,N \equiv M\ \mathbf{to}\ x.\,\lambda y{:}A.\,N$$

**Figure 5** (Typed) equations between CBPV terms.

We define an equational theory for CBPV. We write $\equiv$ for the smallest equivalence relation on terms of the same type that is closed under the axioms in Figure 5 and under the syntax of CBPV terms (for example, $M \equiv N$ implies $\textbf{thunk}\, M \equiv \textbf{thunk}\, N$ and $V \equiv W$ implies $\textbf{return}\, V \equiv \textbf{return}\, W$). (This is not exactly Levy's equational theory for CBPV, because we do not include *complex values*.)

All of the axioms should be read as subject to suitable typing constraints. The group of axioms at the top of Figure 5 contains the $\beta$-laws for all of the type formers except $\mathbf{F}$. The second group contains $\eta$-laws. The bottom group contains axioms governing the behaviour of sequencing of computations: there is a left-unit axiom, a right-unit axiom, an associativity axiom, and axioms for commuting sequencing with the introduction form for functions.