

ARCoSS

LNCS 15695

Viktor Vafeiadis (Ed.)

Programming Languages and Systems

34th European Symposium on Programming, ESOP 2025
Held as Part of the International Joint Conferences on
Theory and Practice of Software, ETAPS 2025
Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part II

2 Part II



 Springer

OPEN ACCESS

Lecture Notes in Computer Science

15695

Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino, USA
Wen Gao, China

Bernhard Steffen , Germany
Moti Yung , USA

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *TU Munich, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen , *University of Dortmund, Germany*
Deng Xiaotie, *Peking University, Beijing, China*
Jeannette M. Wing, *Microsoft Research, Redmond, WA, USA*

More information about this series at <https://link.springer.com/bookseries/558>

Viktor Vafeiadis
Editor

Programming Languages and Systems

34th European Symposium on Programming, ESOP 2025
Held as Part of the International Joint Conferences on
Theory and Practice of Software, ETAPS 2025
Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part II

Editor

Viktor Vafeiadis 

MPI-SWS

Kaiserslautern, Germany



ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-031-91120-0

ISBN 978-3-031-91121-7 (eBook)

<https://doi.org/10.1007/978-3-031-91121-7>

© The Editor(s) (if applicable) and The Author(s) 2025. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

ETAPS Foreword

Welcome to the 28th ETAPS! ETAPS 2025 took place in Hamilton, Canada. It is the first time ETAPS was held outside of Europe.

ETAPS 2025 was the 28th instance of the International Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference established in 1998, and consists of four conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming languages, analysis tools, and formal approaches to software engineering. Organizing these conferences in a coherent, highly synchronized conference programme enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. On the weekend before the main conference, numerous satellite workshops took place that attracted many researchers from all over the globe.

ETAPS 2025 received 329 submissions in total, 106 of which were accepted, yielding an overall acceptance rate of 32.2%. I thank all the authors for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process. Last but not least, my congratulations to all authors of the accepted papers!

ETAPS 2025 featured the unifying invited speakers Ina Schaefer (Karlsruhe Institute of Technology, Germany) and Matthew B. Dwyer (University of Virginia, USA), and the invited speakers Amal Ahmed (Northeastern University, USA) for ESOP and José Meseguer (University of Illinois Urbana-Champaign, USA) for FASE. Invited tutorials were provided by Suguman Bansal (Georgia Institute of Technology, USA) on reinforcement learning from logical specifications and Arun Ross (Michigan State University, USA) on biometrics.

ETAPS 2025 was organized by McMaster University. The Faculty of Engineering at McMaster University has a reputation for innovative programs, cutting-edge research, leading faculty, and aspiring students. It has earned a strong reputation as a center for academic excellence and innovation. The Faculty has approximately 180 faculty members, along with close to 4,500 undergraduate and 1,000 graduate students. The local organization team consisted of Claudio Menghi and Mark Lawford (general chairs), Melissa Alzaeim (event organizer), Alan Wasssyng and Angelo Gargantini (workshop chairs), Sébastien Mosser and Matt Luckcuck (publicity chairs), Patrizio Pelliccione (sponsor chair), Silvia Bonfanti and Andrea Bombarda (web chairs), Jacques Carette and Christos Tsigkanos (local proceedings chair), Lena Liberale and Martin von Mohrenschildt (finance chairs), Damiano Torre and Lina Marsso (registration chairs), and Vera Pantelic and Denise Geiskkovitch (student volunteer chairs).

ETAPS 2025 is further supported by the following associations and societies: ETAPS e.V., EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

The ETAPS Steering Committee consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Marieke Huisman (Twente, chair), Andrzej Wąsowski (Copenhagen), Thomas Noll (Aachen), Jan Kofroň (Prague), Barbara König (Duisburg-Essen), Arnd Hartmanns (Twente), Caterina Urban (INRIA), Jan Křetínský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

Other members of the steering committee are: Elvira Albert (Madrid), Maurice ter Beek (Pisa), Nathalie Bertrand (Rennes), Dirk Beyer (Munich), Artur Boronat (Leicester), Luís Caires (Lisboa), Ferruccio Damiani (Torino), Gordon Fraser (Passau), Arie Gurfinkel (Waterloo), Reiner Hähnle (Darmstadt), Reiko Heckel (Leicester), Marijn Heule (Pittsburgh), Sebastian Junges (Nijmegen), Joost-Pieter Katoen (Aachen and Twente), Guy Katz (Jerusalem), Delia Kesner (Paris), Fabrice Kordon (Paris), Robbert Krebbers (Nijmegen), Kim Guldstrand Larsen (Aalborg), Mark Lawford (Hamilton), Claudio Menghi (Hamilton and Bergamo), Stefan Milius (Erlangen-Nürnberg), Andrzej Murawski (Oxford), Corina Păsăreanu (Ames), Laure Petrucci (Paris), Peter Y.A. Ryan (Luxembourg), Don Sannella (Edinburgh), Viktor Vafeiadis (Kaiserslautern), and Anton Wijs (Eindhoven).

I would like to take this opportunity to thank all authors, keynote speakers, attendees, organizers of the satellite workshops, and Springer Nature for their support. ETAPS 2025 was also generously supported by Tourism Hamilton and the Tutte Institute for Mathematics and Computing. I hope you all enjoyed ETAPS 2025.

Finally, a big thanks to Claudio, Mark and Melissa and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

May 2025

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

The LNCS volumes 15694 and 15695 contain papers that were presented at the 34th European Symposium on Programming (ESOP 2024), held during May 3–8 in Hamilton, Canada. ESOP is part of the International Joint Conferences on Theory and Practice of Software (ETAPS) and promotes the specification, design, analysis and implementation of programming languages and systems.

In total, these two volumes contain 30 papers and two short artifact reports associated with the corresponding research papers. Following ESOP 2024, ESOP 2025 had three submission categories: research papers, “fresh perspective” papers and experience report. For the first time in its history, ESOP 2025 also instituted two submission and evaluation rounds. The first round had a submission deadline on 30th May 2024, while the second had the usual submission deadline in October. Round 1 submissions could receive one of three outcomes: Accept, Reject, Revise. In the latter case, the PC provided a concrete list of revision requests to be completed by the Round 2 submission deadline. Such revised submissions were then reviewed by the same set of reviewers and were either accepted or rejected. Rejected Round 1 submissions were prevented from being resubmitted to Round 2. Papers submitted directly to Round 2 were either accepted or rejected.

The first round attracted 26 submissions, of which 8 (31%) were accepted, and 13 were rejected completely and another 5 were rejected but allowed to resubmit a revised version to the second submission deadline. The second deadline attracted 67 submissions, 5 of which were revisions of rejected round-1 submissions. Of these submissions, 22 papers (33%) were accepted: 4 of the 5 round-1 resubmissions, and 18 of the 62 fresh submissions. In total, this amounts to 30 accepted papers out of 88 unique submissions, yielding an acceptance rate of 34%. Or, if accounted differently, 30 accepted papers out of 93 total submissions, yielding an acceptance rate of 32%. By submission category, 27 of the 76 research papers were accepted (36%), 2 of the 9 fresh perspective papers were accepted (22%), 1 of the 3 experience reports was accepted (33%).

Papers were reviewed in a double-blind fashion, with author identities only revealed to reviewers on paper acceptance. Each submission typically received three reviews by the members of the ESOP program committee, occasionally supported by external reviewers. In both submission rounds, authors were given a chance to respond to their reviews, before the program was selected through an online, asynchronous PC meeting, facilitated by the HotCRP system.

ESOP 2025 also employed an artifact evaluation process. Fourteen of the accepted papers elected to make their artifacts available on archival websites. The committee awarded the badge “Functional” to seven of these and the badges “Functional and reusable” to the remaining seven. Two accepted papers in this volume are accompanied by artifact reports.

My sincere thanks go to all who worked together to produce this event and its proceedings. Foremost, to the authors, who provided the technical content of the meeting. Also to the program committee, artifact evaluation committee, and external reviewers,

who provided their well-reasoned and detailed judgments, sometimes on short notice. Michalis Kokologiannakis, as the representative for ESOP among the artifact evaluation committee co-chairs, deserves particular thanks. I also would like to thank the ETAPS steering committee and its chair Marieke Huisman, the Proceedings coordinator Barbara König, the local proceedings chairs Jacques Carette and Christos Tsigkanos, and the webmaster Jan Kofron for their assistance in the organization of ESOP as a part of the entire ETAPS meeting. Finally, thanks are due to the members of the ESOP steering committee and, in particular, its chair, Luís Caires, who were a constant source of support, encouragement, information and guidance.

May 2025

Viktor Vafeiadis

Organization

ESOP Steering Committee

Luís Caires	Instituto Superior Técnico, Universidade de Lisboa, Portugal
Robbert Krebbers	Radboud University Nijmegen, Netherlands
Brigitte Pientka	McGill University, Canada
Thomas Wies	New York University, USA
Nobuko Yoshida	Imperial College London, UK
Viktor Vafeiadis	MPI-SWS, Germany
Stephanie Weirich	University of Pennsylvania, USA

ESOP 2025 Program Chair

Viktor Vafeiadis	MPI-SWS, Germany

ESOP Program Committee

Kazuyuki Asada	Tohoku University, Japan
Dariusz Biernacki	University of Wrocław, Poland
Laura Bocchi	University of Kent, UK
James Cheney	University of Edinburgh, UK
Brijesh Dongol	University of Surrey, UK
Jana Dunfield	Queen's University at Kingston, Canada
Javier Esparza	Technical University of Munich, Germany
Simon Fowler	University of Glasgow, UK
Jacques-Henri Jourdan	CNRS, France
Burcu Kulahcioglu Ozkan	Delft University of Technology, Netherlands
Hongjin Liang	Nanjing University, China
Umay Mathur	National University of Singapore, Singapore
Peter Müller	ETH Zürich, Switzerland
David Pichardie	Meta, France
Jean Pichon	Aarhus University, Denmark
Noam Rinetzky	Tel Aviv University, Israel
Amr Sabry	Indiana University, USA
Michael Sammler	ETH Zürich, Switzerland

Sam Staton	University of Oxford, UK
Milijana Surbatovich	University of Maryland, College Park, USA
Niki Vazou	IMDEA Software Institute, Spain
Tobias Wrigstad	Uppsala University, Sweden

ESOP/FASE/FoSSaCS Joint Artifact Evaluation Committee

AEC Co-chairs

Michalis Kokologiannakis	ETH Zürich, Switzerland
Laura Bussi	University of Luxembourg, Luxembourg
Stefan Winter	LMU Munich, Germany
Ondřej Lengál	Brno University of Technology, Czech Republic

AEC Members

Alexandre Moine	NYU, USA
András Kovács	University of Gothenburg, Sweden
Andrea Colledan	University of Bologna, Italy
Bernardo Almeida	LASIGE, University of Lisbon, Portugal
David Chocholatý	Brno University of Technology, Czech Republic
Gennaro Zanfardino	University of L'Aquila, Italy
Giordano d'Aloisio	University of L'Aquila, Italy
Gustavo Carvalho	Universidade Federal de Pernambuco, Brazil
Hongjian Jiang	RPTU, Germany
Julia Sapiña	Universitat Politècnica de València, Spain
Loïc Pujet	Stockholm University, Sweden
Loïc Germerie Guizouarn	Université de Rennes, CNRS, Inria, IRISA, France
Lucas Sakizoglou	Brandenburg University of Technology, Germany
Manolis Pitsikalis	NCSR Demokritos, Greece
Michal Hečko	Brno University of Technology, Czech Republic
Noa Izsak	Ben-Gurion University of the Negev, Israel
Pablo Gómez-Abajo	Universidad Autónoma de Madrid, Spain
Raúl Gutiérrez	Universitat Politècnica de València, Spain
Raúl López-Rueda	Universitat Politècnica de València, Spain
Sougata Bose	University of Liverpool, UK
Soumodev Mal	Chennai Mathematical Institute, India
Srinidhi Nagendra	IRIF, CNRS, Université Paris Cité, France, and Chennai Mathematical Institute, India

Szumi Xie
Thomas Holger
Vincent Cheval
Wei-Lun Tsai
Zainab Fatmi
Zsófia Ádám

Eötvös Loránd University, Hungary
Forschungsinstitut CODE, Germany
University of Oxford, UK
Academia Sinica, Taiwan
University of Oxford, UK
Budapest University of Technology and
Economics, Hungary

Contents – Part II

Efficient Synthesis of Tight Polynomial Upper-Bounds for Systems of Conditional Polynomial Recurrences	1
<i>Amir K. Goharshady, S. Hitarth, and Sergei Novozhilov</i>	
CUTECat: Concolic Execution for Computational Law	31
<i>Pierre Goutagny, Aymeric Fromherz, and Raphaël Monat</i>	
First-Person Choreographic Programming with Continuation-Passing Communications	62
<i>Sung-Shik Jongmans</i>	
Formal Verification of WTO-based Dataflow Solvers	91
<i>Roméo La Spina, Delphine Demange, and Sandrine Blazy</i>	
Formal Verification of WTO-based Dataflow Solvers: Artifact Experience Report	119
<i>Roméo La Spina, Delphine Demange, and Sandrine Blazy</i>	
Multiparty Session Types with a Bang!	125
<i>Matthew Alan Le Brun, Simon Fowler, and Ornela Dardha</i>	
Formal Autograding in a Classroom	154
<i>Dragana Milovančević, Mario Bucev, Marcin Wojnarowski, Samuel Chassot, and Viktor Kunčák</i>	
COMA, an Intermediate Verification Language with Explicit Abstraction Barriers	175
<i>Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre</i>	
A Complete Axiomatisation of Equivalence for Discrete Probabilistic Programming	202
<i>Robin Piedeleu, Mateo Torres-Ruiz, Alexandra Silva, and Fabio Zanasi</i>	
Compositional Shape Analysis with Shared Abduction and Biabductive Loop Acceleration	230
<i>Florian Sextl, Adam Rogalewicz, Tomáš Vojnar, and Florian Zuleger</i>	
SMT-Boosted Security Types for Low-Level MPC	258
<i>Christian Skalka and Joseph P. Near</i>	

Context-Dependent Effects in Guarded Interaction Trees 286
Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany,
and Lars Birkedal

An Automata-theoretic Basis for Specification and Type Checking
of Multiparty Protocols 314
Felix Stutz and Emanuele D’Oswaldo

Named Arguments as Intersections, Optional Arguments as Unions 347
Yaozhu Sun and Bruno C. d. S. Oliveira

Context-Sensitive Demand-Driven Control-Flow Analysis 374
Tim Whiting and Kimball Germane

On the Relationship between Dijkstra Monads and Higher-Order Fixpoint
Logic 402
Risa Yamada, Naoki Kobayashi, Ken Sakayori, and Ryosuke Sato

Author Index 429

Contents – Part I

The Vanilla Sequent Calculus is Call-by-Value	1
<i>Beniamino Accattoli</i>	
Formulas as Processes, Deadlock-Freedom as Choreographies	23
<i>Matteo Acclavio, Giulia Manara, and Fabrizio Montesi</i>	
Sufficient Conditions for Robustness of RDMA Programs	56
<i>Guillaume Ambal, Ori Lahav, and Azalea Raad</i>	
Constructive characterisations of the MUST-preorder for asynchrony	88
<i>Giovanni Bernardi, Iliaria Castellani, Paul Laforgue, and Léo Stefanescu</i>	
Abstraction of memory block manipulations by symbolic loop folding	117
<i>Jérôme Boillot and Jérôme Feret</i>	
Cognacy Queries over Dependence Graphs for Transparent Visualisations	144
<i>Joe Bond, Cristina David, Minh Nguyen, Dominic Orchard, and Roly Perera</i>	
An abstract, certified account of operational game semantics	172
<i>Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski</i>	
Artifact Report: an Abstract, Certified Account of Operational Game Semantics	200
<i>Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski</i>	
Neural Network Verification is a Programming Language Challenge	206
<i>Lucas C. Cordeiro, Matthew L. Daggitt, Julien Girard-Satabin, Omri Isac, Taylor T. Johnson, Guy Katz, Ekaterina Komendantskaya, Augustin Lemesle, Edoardo Manino, Artjoms Šinkarovs, and Haoze Wu</i>	
Stratified Type Theory	236
<i>Jonathan Chan and Stephanie Weirich</i>	
Coverage Semantics for Dependent Pattern Matching	264
<i>Joseph Eremondi and Ohad Kammar</i>	
Variable Elimination as Rewriting in a Linear Lambda Calculus	292
<i>Thomas Ehrhard, Claudia Faggian, and Michele Pagani</i>	

A Program Logic for Concurrent Randomized Programs in the Oblivious
Adversary Model 322
Weijie Fan, Hongjin Liang, Xinyu Feng, and Hanru Jiang

Iso-Recursive Multiparty Sessions and their Automated Verification 349
Marco Giunti and Nobuko Yoshida

Verifying Algorithmic Versions of the Lovász Local Lemma 379
Rongen Lin, Hongjin Liang, and Xinyu Feng

Elucidating Type Conversions in SQL Engines 408
*Wenjia Ye, Matías Toro, Claudio Gutierrez, Bruno C. d. S. Oliveira,
and Éric Tanter*

Author Index 437



Efficient Synthesis of Tight Polynomial Upper-bounds for Systems of Conditional Polynomial Recurrences

Amir K. Goharshady¹(✉), S. Hitarth², and Sergei Novozhilov²

¹ University of Oxford, Oxford, UK

`amir.goharshady@cs.ox.ac.uk`

² Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
`{hsinghab,snovozhilov}@connect.ust.hk`

Abstract. Recurrence relations are used in a wide variety of static program analysis tasks such as loop summarization, invariant generation, and, most classically, modeling the (asymptotic) worst-case runtime behavior of recursive divide-and-conquer algorithms. In this work, we focus on the latter use-case. Classical methods for this problem, such as the well-known Master Theorem (MT) or the Akra-Bazzi method (AB) can only handle *single* recurrences of a certain limited form. Specifically, MT requires that each instance be divided into a *fixed* number of smaller sub-instances of the *same size*. AB generalizes MT by allowing sub-instances of different sizes, but still requires that the number of such sub-instances be fixed and independent of the size of the original instance. Moreover, these methods can handle neither *multi-variate* recurrences nor *systems* of recurrences that model non-simple recursive behavior among two or more procedures.

In this work, we lift these restrictions and consider a wide family of recurrences called Generalized Polynomial Recurrence Systems (GPRS). Our setting is highly expressive and allows systems of multi-variate recurrences in which an instance can be divided into polynomially-many sub-instances. Moreover, the division is not limited to a single rule and can have several cases based on conditions on the values of the variables. We show how to obtain polynomial upper-bounds for a GPRS using template-based methods and classical theorems from polyhedral and algebraic geometry. Our approach reduces the synthesis of polynomial upper-bounds to linear or semi-definite programming instances, enabling efficient solutions. Crucially, our method is sound and semi-complete, i.e. complete for polynomials of any fixed degree and obtains concrete, as opposed to asymptotic, upper-bounds.

Keywords: Recurrence Relations · Positivstellensätze · Automated Verification.

1 Introduction

RECURRENCE RELATIONS. A *recurrence relation* is any equation expressing the n -th element of a sequence in terms of its previous elements. A *system of*

recurrence relations is a family of recurrence relations for multiple sequences, where the n -th element of each sequence is expressed in terms of previous elements of not only that sequence, but also the other sequences in the family. For example, consider the following system: $x_n = 2 \cdot x_{n-1} + y_{n-1} \wedge y_n = x_{n-1} + 2 \cdot y_{n-1}$. This is an example of a system of *linear recurrences*, since each element is a linear combination of previous elements.

RECURRENCE RELATIONS FOR RUNTIME ANALYSIS. Recurrence relations are the most classical and standard formalism to capture the worst-case runtime behavior of recursive or divide-and-conquer algorithms with respect to the size n of their input [38]. When analyzing the asymptotic worst-case runtime of a recursive algorithm, the goal is to obtain an *asymptotic upper-bound* on the value of the n -th term of the recurrence. The two classical approaches here are the master theorem (MT) and the Akra-Bazzi method (AB).

MASTER THEOREM (MT). First introduced in 1980 [12], the master theorem considers recurrences of the form $T(n) = a \cdot T(n/b) + f(n)$ and provides a recipe-like formula for an asymptotic solution in several cases with respect to a, b and f . For example, the solution is $O(n^{\log_b a})$ when $f(n) \in O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$ and $O(n^{\log_b a} \cdot \log n)$ when $f(n) \in \Theta(n^{\log_b a})$. Given its ability to easily obtain bounds on the runtime of many common algorithms, MT is often included in undergraduate algorithms courses and textbooks [38].

AKRA-BAZZI METHOD (AB). A significant limitation in MT is that it can only handle divide-and-conquer algorithms in which every instance is divided into a fixed number a of sub-instances of the same size n/b . The Akra-Bazzi method [1] extends MT by allowing sub-instances of differing sizes. Specifically, it handles recurrences of the form

$$\forall n \geq n_0 \quad T(n) = f(n) + \sum_{i=1}^k a_i \cdot T(b_i \cdot n + g_i(n)),$$

in which

- (1) $a_i > 0$ and $0 < b_i < 1$ are constants for each i .
- (2) $|f'(n)| \in O(n^c)$ for a fixed constant c and $g_i(n) \in O(n/\log^2 n)$.

For such recurrences, AB first finds a value p such that $\sum_{i=1}^k a_i \cdot b_i^p = 1$ and then proves the asymptotic upper-bound

$$T(n) \in O\left(n^p \cdot \left(1 + \int_1^n g(x)/x^{p+1} dx\right)\right).$$

Both MT and AB can only handle linear recurrences.

RECURRENCE SOLVING IN STATIC ANALYSIS. Recurrence relations are also utilized as a way to summarize loops [75,47,70] and generate invariants [74,72,15]. Invariant generation is an essential first step in a wide variety of static program analyses [85,76,64]. Similarly, loop summarization enables the application of powerful methods such as algebraic [71,31,34] and Newtonian [84,45] program analysis. A crucial difference between this kind of recurrence solving and the one in MT/AB is that worst-case runtime analysis aims to obtain an *upper-bound* on the n -th element $T(n)$ of the recurrence. This upper-bound is often asymptotic, i.e. ignoring constant factors. In contrast, loop summarization

and invariant generation methods need an exact closed-form formula for the n -th element. This leads to a harder problem, especially since the recurrence might not have a closed-form solution, and thus the tradeoff for these approaches is that they often handle a more restricted family of recurrences. In this work, we follow MT and AB and focus on finding upper-bounds, but our upper-bounds are not asymptotic and do not ignore constant factors.

SYSTEMS OF RECURRENCES. A major limitation in AB and MT is that they can only handle a single recurrence relation. However, it is easy and natural to write programs consisting of several procedures with non-trivial recursive calls between them. In these cases, we will have several sequences $T_1, T_2, \dots, T_k$, each corresponding to the runtime of one of the procedures, and the n -th element in each of them can depend on previous elements in the other sequences, as well.

MULTI-VARIATE RECURRENCES. Another limitation of MT and AB is that they only consider recurrences with a single variable n , which is intuitively a measure of the size of the instance passed to the underlying recursive algorithm. However, there are many cases in which the size of an instance is not adequately captured by a single parameter. For example, the runtime of an algorithm that works with graphs is often dependent on both the number of vertices $|V|$ and the number of edges $|E|$ in the input graph. If we simply merge the two and define $n := |V| + |E|$, this might lead to a gross over-approximation of the actual worst-case runtime of the algorithm. Indeed, the entire field of parameterized algorithms and complexity [39,44] is based on studying the worst-case runtime of algorithms using more than one parameter. In such cases, MT and AB are not applicable and the runtimes are analyzed manually.

CONDITIONAL RECURRENCE RELATIONS. A further challenge in solving recurrences is that many applications lead to so-called *conditional* recurrence relations [91]. These are recurrences in which there is no fixed formula for each element of the sequence, but we instead have several formulas and different conditions for the application of each. As a simple example, we might have two parameters n and m and different formulas for $T(n, m)$ depending on whether n is larger than m or not. In such cases, neither AB nor MT can be applied. The state-of-the-art for conditional recurrences is [91] which can only handle periodic linear recurrences.

PROBABILISTIC RECURRENCE RELATIONS. Another related family of works are those that consider probabilistic recurrence relations, thus bounding the expected runtime of a randomized recursive algorithm such as Quick Sort or the probability that its runtime exceeds a certain threshold. In their setting, the size of the sub-instances is not necessarily deterministic but can instead be probabilistic. However, the number of sub-instances is fixed. For example, Quick Sort breaks down an array of size n into two arrays of size i and $n - i - 1$, where each value of i is equally likely. The standard approach in this area is known as Karp’s cookbook [69,68]. and resembles MT. Extensions of Karp’s cookbook include [35,90,89].

SKETCH OF OUR APPROACH. In this work, we consider systems of multi-variate recurrence relations, in which the number of sub-instances is not neces-

sarily fixed. Our approach is template-based. Before getting into formal details in the coming sections, we first illustrate the main ideas of our method using a simple example.

- Consider the following toy recurrence relation:

$$x_0 = 1 \quad \forall n \geq 1, \quad x_n = x_{n-1} + n.$$

Using repeated substitution, we can obtain $x_n = 1 + 1 + 2 + \dots + n = 0.5 \cdot n^2 + 0.5 \cdot n + 1$. We would like to find upper-bounds automatically and without such manual analysis.

- Suppose $f(n) := a \cdot n^2 + b \cdot n + c$ is a polynomial such that $f(n) \geq f(n-1) + n$ when $n \geq 1$ and $f(0) = 1$. It is straightforward to see that $f(n)$ will be a valid upper-bound on x_n . Our goal is to find concrete values for a, b, c .
- Thus, our problem is reduced to finding values for a, b, c that witness the satisfiability of the following formula in the first-order theory of the reals:

$$\exists a, b, c \quad \forall n \quad (n \geq 1 \Rightarrow f(n) \geq f(n-1) + n) \wedge f(0) = 1.$$

With a simple substitution we obtain:

$$\exists a, b, c \quad \forall n \quad (n \geq 1 \Rightarrow (2 \cdot a - 1) \cdot n - a + b \geq 0) \wedge c = 1 \quad (1)$$

While this particular example is easy to solve, the formula at this point would generally contain polynomials of higher degrees. It is a formula in the first-order theory of the reals, which also has an $\exists\forall$ quantifier alternation. Thus, a direct solution is not viable even for small instances [11, 77, 18].

- Depending on the degree of polynomials appearing in the constraint, we use various theorems from polyhedral or real algebraic geometry to eliminate the quantifier alternation and reduce our problem to either linear programming (LP) or semi-definite programming (SDP). Both LP and SDP are known to be PTIME-solvable and there are many practically scalable tools able to handle large LP and SDP instances. Here, we know we have to set $c = 1$, leading to

$$\exists a, b \quad \forall n \quad (n - 1 \geq 0 \implies (2 \cdot a - 1) \cdot n - a + b \geq 0).$$

Since all the polynomials involved are linear, we apply the well-known Farkas' lemma [46].

- The idea behind Farkas' lemma is to try to write the RHS of the implication as a linear combination with non-negative coefficients of $1 \geq 0$ and the LHS. In other words, we would like to find $\lambda_0, \lambda_1 \geq 0$ such that

$$\forall n \quad (2 \cdot a - 1) \cdot n - a + b = \lambda_0 + \lambda_1 \cdot (n - 1).$$

Since the polynomial identity above should hold for all values of n , we must have the same corresponding coefficients, i.e. the constant term and the coefficient of n should be equal on both sides of the equation. Thus, we obtain $2 \cdot a - 1 = \lambda_1$ and $-a + b = \lambda_0 - \lambda_1$. We also had $c = 1$ from

before. Hence, our problem is now reduced to an LP instance over a, b, c, λ_0 and λ_1 . It is now easy to verify that $a = b = 0.5, c = 1, \lambda_0 = \lambda_1 = 0$ is a solution. Plugging the values for a, b, c back into the template for f , we obtain $f(n) = 0.5 \cdot n^2 + 0.5 \cdot n + 1$, which is a tight polynomial bound on x_n . Crucially, every step of this algorithm is entirely automated and does not require any manual analysis.

OUR CONTRIBUTION. In this work, we define an expressive class of recurrences called Generalized Polynomial Recurrence Systems (GPRS). Our GPRS setting is more expressive than that of MT and AB in the following dimensions:

- We allow *multi-variate* recurrences.
- We handle *systems* of recurrences instead of a single one.
- The recurrence relations need not be linear and can instead be polynomial. For example, we can have $T(n) = p(n) \cdot T(q(n)) + r(n)$ where p, q and r are polynomials³.
- Our recurrence relations can be *conditional*.

We present the following results based on GPRS:

- *Decidability.* We prove that the problem of finding a polynomial upper-bound of any fixed degree d is decidable, assuming the GPRS represents a well-defined family of positive sequences. This is achieved by a reduction of the problem to a formula in the first-order theory of the reals, similar to (1). In Theorem 1, we prove that this reduction is both sound and complete. We then apply the well-known Tarski-Seidenberg theorem [87,86] for decidability.
- *Template-based Algorithm.* Formulas in the first-order theory of the reals are notoriously hard to decide, especially when they involve quantifier alternation. The general techniques for such constraints take doubly-exponential time [52]. To avoid this, we utilize several theorems from polyhedral and algebraic geometry, including Farkas' lemma [46], Handelman's Theorem [51] and Putinar's Positivstellensatz [83], to eliminate the quantifier alternation and reduce the problem to either linear programming (LP) or semi-definite programming (SDP). There are many modern scalable solvers for LP/SDP and both are in PTIME.
- *Completeness and Tightness.* Assuming that the GPRS has a polynomial upper-bound of the desired degree, and that polynomial templates of high enough degree are used in the algorithm, our approach is guaranteed to find such a polynomial upper-bound. Moreover, every upper-bound polynomial corresponds to a unique solution of the LP/SDP instance. Thus, our approach can find the tightest possible bound with respect to any user-defined objective.

In short, we provide automated sound and complete algorithms that, for any fixed degree of the templates, can find the tightest polynomial bound for a given

³ However, our setting is more restrictive in the sense that all parts of the recurrence relation have to be polynomial, including the time $r(n)$ spent on preprocessing each instance.

GPRS by means of a reduction to LP/SDP. As argued above, our setting is more general than the classical methods of MT and AB in several dimensions. We also provide an extensive set of illustrative examples, showcasing that our approach is applicable to recurrence relations that cannot be handled by MT or AB.

LIMITATIONS. The main limitation of our approach is that it requires everything to be polynomial. More specifically, all parts of the input GPRS, as well as the synthesized upper-bound have to be polynomial. This is a natural and inherent limitation due to our dependence on theorems in real algebraic geometry, which is the study of systems of polynomials and their real solutions. This being said, we can still apply the classical techniques of *polynomial substitution* and *logarithmic substitution* [38], to obtain exponential bounds such as 3^n or bounds that have rational non-integral degrees, such as $n^{2.83}$. We provide examples of such bounds in Sections 6 and 7. Moreover, we focus on the problem of solving systems of recurrence relations and do not consider how the system was obtained in the first place. This is natural in our main use-case, i.e. worst-case runtime analysis of algorithms, in which there is no implementation of the algorithm and instead the algorithm designer manually finds the recurrences and uses them to analyze asymptotic/exact runtimes [48]. In comparison, there are several previous works, such as [65,15,70], that extract systems of recurrences from programs. For such scenarios, our algorithm can only be applied to find the tightest possible polynomial bound of a given degree after the system of recurrences is generated by a separate tool.

STELLENSATZ-BASED PROGRAM ANALYSIS. Our approach is a template-based algorithm using Farkas’ Lemma, Handelman’s Theorem and Putinar’s Positivstellensatz. These three theorems are classical and well-known results in polyhedral and real algebraic geometry. They have also been used in several previous works in static analysis, especially for invariant generation [36,26], quantifier elimination [3], LTL model-checking [29] and program synthesis [50,2]. These works reduce the respective static analysis problems to Quadratically-Constrained Quadratic Programming (QCQP) and rely on the Real Nullstellensatz for their completeness. In contrast, we do not use the Nullstellensatz and our reduction is to Linear Programming (LP) or Semi-definite Programming (SDP). QCQP is an *NP-hard* problem whereas both LP and SDP have PTIME solutions. Thus, for any fixed degree of template polynomials, our approach takes *polynomial time*. This indicates that, unsurprisingly, template-based recurrence solving is an easier problem in terms of complexity than both invariant generation and program synthesis. Similar approaches have also been used in the context of termination analysis, e.g. for the synthesis of ranking functions and ranking supermartingales [23,82,28,24,27,25,63,93,32]. For any fixed degree of template polynomials, if tight program invariants are given as part of the input, then these approaches find ranking functions/supermartingales in PTIME. However, generating the required invariants themselves is not PTIME as mentioned above. We also remark that our algorithms do not depend on invariants or any input other than the system of recurrences.

OTHER RECURRENCE SOLVING APPROACHES. The closest previous work is a template-based recurrence solving approach and tool called CHORA [15], which builds upon [47]. In our experiments (Section 7), the outputs of the two approaches often coincide. However, in contrast to our approach, CHORA does not provide any completeness guarantees. Thus, there are instances in which our approach is able to synthesize a bound but CHORA fails. There are also instances in which our approach’s polynomial bound is tighter than CHORA’s. On the other hand, CHORA’s setting is more general since we focus only on polynomial templates and solutions, whereas CHORA creates templates in which arbitrary functions can serve as unknowns. For example, it is able to find a non-polynomial bound of $O(n \cdot \lg n)$ for merge sort. Therefore, neither approach is subsumed by the other. A recent related work is [40], which also focuses on generality, i.e. being able to soundly synthesize bounds of different forms, rather than completeness for a particular type of bound. It takes a program as input and applies abstractions to enable the application of algebraic program analysis [71,30,37,33] and methods for generating all polynomial invariants. Note that, as mentioned above, this is a strictly harder problem than the one considered in our setting. Another related work is [70], which automatically infers recurrences from programs and then provides a highly efficient compositional method to solve them. This method is efficient but not complete. An interesting direction of future work would be to combine [70] with our approach, i.e. first use [70] to obtain recurrence systems from programs and then use a combination of our complete method for polynomial bounds and [70]’s method for non-polynomial bounds to solve the resulting recurrences. Finally, there are many recurrence-solving methods developed as subroutines in the context of automated (amortized) resource analysis [59,60,61,62,7,17,19,20,94,21,16,22,41,42,43,53,56,57,66,67,92]. Similar to the works above, these approaches consider a program directly, extract recurrence relations modeling its resource usage and then use sound but incomplete solution methods based on a variety of techniques, such as type systems or separation logic, for general non-polynomial bounds. See [58] for an excellent survey.

2 Polynomial Recurrence Systems

In this work, we use a formalism called Generalized Polynomial Recurrence Systems (GPRS), which only employs polynomials in the recurrence relations. We allow recurrence relations such as $T(n) = 2 \cdot T(n/2) + n^2$ but not $T(n) = 2 \cdot T(n/2) + 2^n$. In this section, we formally define the concept of a GPRS, its syntax and its semantics.

RECURRENCE VARIABLES AND VALUATIONS. We assume the recurrence relations are defined over a finite set of variables $\mathbf{n} = \{n_1, \dots, n_k\}$, called *recurrence variables*. The set of polynomials in the variables $\mathbf{n}$ over the reals is denoted by $\mathbb{R}[\mathbf{n}]$. The domain of our recurrence variables is the set $\mathbb{R}^+$ of non-negative real numbers. A *valuation* is a map $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$. We evaluate a polynomial $p(\mathbf{n}) \in \mathbb{R}[\mathbf{n}]$ at a valuation σ by substituting each variable n with the value $\sigma(n)$

and computing p with the usual meaning of multiplication and addition. This evaluation is denoted by either $\sigma(p(\mathbf{n}))$ or $p(\sigma(\mathbf{n}))$.

RECURRENCE SYMBOLS AND POLYNOMIAL CALLS. A *recurrence symbol* over $\mathbf{n}$ is simply a function symbol $T(\mathbf{n})$. We denote the set of all recurrence symbols by $\mathcal{T}$. A *polynomial substitution map* $\delta : \mathbf{n} \rightarrow \mathbb{R}[\mathbf{n}]$ is a function that assigns a polynomial $p(\mathbf{n}) \in \mathbb{R}[\mathbf{n}]$ to each variable $n \in \mathbf{n}$. A polynomial substitution map can be applied to a recurrence symbol to get $\delta(T(\mathbf{n})) = T(\delta(\mathbf{n}))$. We call $T(\delta(\mathbf{n}))$ the *polynomial call* of $T(\mathbf{n})$ with respect to δ . For example, let $T(n, m)$ be a recurrence symbol over $\mathbf{n} = \{n, m\}$ and δ a polynomial substitution map with $\delta(n) = n + m$ and $\delta(m) = m - 1$. The polynomial call of $T(n, m)$ with respect to δ is $\delta(T(n, m)) = T(\delta(n), \delta(m)) = T(n + m, m - 1)$.

SYNTAX. A *Generalized Polynomial Recurrence System* (GPRS) is generated from the non-terminal S in the grammar of Figure 1. To make the system more human-readable, we also explicitly write the recurrence variables and symbols at the top of the system as shown in Figure 2.

$$\begin{array}{ll}
 S := \phi; S \mid \phi & \phi := \text{if } (\varphi) \text{ then } T(\mathbf{n}) = \mathcal{R} \\
 \varphi := f \geq 0 \mid \varphi \wedge \varphi & \mathcal{R} := \mathcal{R} + \mathcal{R} \mid f \cdot \mathcal{M} \mid f \\
 \mathcal{M} := T(\delta(\mathbf{n})) & \delta : \mathbf{n} \rightarrow \mathbb{R}[\mathbf{n}] \\
 T \in \mathcal{T} & f \in \mathbb{R}[\mathbf{n}]
 \end{array}$$

Fig. 1: Our Syntax

Example 1. An example GPRS is given in Figure 2. In line 1, we define that our recurrences will use two variables n and m . In line 2, we define a recurrence symbol T over these variables. Finally, in lines 3 and 4, we describe the equations of our recurrence using **if-then** blocks. Our grammar allows a system of multiple recurrences containing arbitrary polynomials over the recurrence variables.

```

@var: n, m; @recurrence: T(n, m);
if (m ≥ 1) then T(n, m) = T(n + m, m - 1) + m
if (n ≥ 0 ∧ m = 0) then T(n, m) = n

```

Fig. 2: An Example GPRS

MONOMIALS AND COEFFICIENTS. An $\mathcal{M}$ -monomial is any expression generated from the non-terminal $\mathcal{M}$ in the grammar of Figure 1, and an $\mathcal{R}$ -expression R is any expression generated from the non-terminal $\mathcal{R}$. If an $\mathcal{R}$ -expression is of the form $R := f \cdot M + R'$, where M is an $\mathcal{M}$ -monomial, then f is said to be

the *coefficient* of M . We can evaluate an $\mathcal{R}$ -expression R at any valuation σ by applying the substitution $n \rightarrow \sigma(n)$ in R and simplifying it. We denote this evaluation by $\sigma(R)$.

GPRS NOTATION. Based on the discussion above, let $\mathcal{T}$ be the set of recurrence symbols and $\mathbf{n}$ the set of recurrence variables. A **GPRS** $\mathcal{F}(\mathcal{T}, \mathbf{n}) = \{\phi_1, \dots, \phi_m\}$ is a set of **if-then** blocks with each $\phi_i := \text{if } (\varphi_i) \text{ then } T_i(\mathbf{n}) = R_i$ where each R_i is an $\mathcal{R}$ -expression. ϕ_i .

SEMANTICS. Let $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$ be a valuation. Given a GPRS $\mathcal{F}$, we interpret each $\phi \in \mathcal{F}$ as a *conditional rewriting rule* for $T(\mathbf{n})$. We call $G(\phi) := \varphi$ the *guard* of ϕ and $\mathcal{R}(\phi) := (T(\mathbf{n}) = R)$ the *rewrite rule* of ϕ . If the valuation σ satisfies the guard $G(\phi)$, we can invoke the rewrite rule $\mathcal{R}(\phi)$ of ϕ on $T(\sigma(\mathbf{n}))$ and rewrite $T(\sigma(\mathbf{n})) \rightarrow \sigma(R)$. A rewrite $R \rightarrow R'$ such that $R' \neq R$ is called *non-trivial*.

To be more formal, we define the application of a conditional rewrite rule $\phi := \text{if } (\varphi) \text{ then } G(\mathbf{n}) := R$ on an $\mathcal{R}$ -expression by structural induction:

$$\begin{aligned} \phi[\mathcal{R} + \mathcal{R}] &:= \phi[\mathcal{R}] + \phi[\mathcal{R}] \\ \phi[f \cdot \mathcal{M}] &:= \phi[f] \cdot \phi[\mathcal{M}] \\ \phi[f(\sigma(\mathbf{n}))] &:= f(\sigma(\mathbf{n})) \\ \phi[T(\sigma(\delta(\mathbf{n}))) &:= \begin{cases} \sigma(R), & \text{if } T = G \text{ and } \sigma \models \varphi \\ T(\sigma(\delta(\mathbf{n}))), & \text{otherwise} \end{cases} \end{aligned}$$

EVALUATIONS. An *evaluation* of an $\mathcal{R}$ -expression R , such as $T(\mathbf{n})$, at a valuation σ in the GPRS $\mathcal{F}$ is a finite sequence $\mathcal{E} := R_0 \xrightarrow{\phi_1} R_1 \xrightarrow{\phi_2} \dots \xrightarrow{\phi_n} R_n$ of non-trivial rewrites such that $R_0 := \sigma(R)$, $R_i = \phi_i[R_{i-1}]$ for each $i \geq 1$, and R_n is a numerical expression which does not contain any non-terminals and thus can be evaluated to a real number. $\mathcal{E}(\sigma(R)) := R_n$ is called the *value* of $\sigma(R)$ under the evaluation $\mathcal{E}$. The evaluations are not necessarily unique and we do allow non-determinism. A GPRS $\mathcal{F}$ is *non-deterministic* if there are two rules $\phi, \psi \in \mathcal{F}$ for $T(\mathbf{n})$ and a valuation σ such that $\sigma \models \phi$ and $\sigma \models \psi$.

Example 2. Consider the GRPS given in Example 1. It consists of the following two rewrite rules:

$$\begin{aligned} \phi_1 &:= \text{if } (m \geq 1) \text{ then } T(n, m) = T(n + m, m - 1) + m \\ \phi_2 &:= \text{if } (n \geq 0 \wedge m = 0) \text{ then } T(n, m) = n \end{aligned}$$

Take the valuation $n \mapsto 5, m \mapsto 2$. An evaluation of $T(5, 2)$ is $T(5, 2) \xrightarrow{\phi_1} T(7, 1) + 2 \xrightarrow{\phi_1} T(8, 0) + 1 + 2 \xrightarrow{\phi_2} 8 + 1 + 2 = 11$.

Example 3. Consider the following GPRS:

$$\text{@var: } n; \text{ @recurrence: } T(n); \text{ if } (n \geq 1) \text{ then } T(n) = T(n + 1) + n$$

If we try to evaluate $T(2)$ by rewriting, we get $T(2) \rightarrow T(3) + 2 \rightarrow (T(4) + 3) + 2 \rightarrow \dots$ and the sequence never ends. Indeed, it is impossible to evaluate $T(n)$ for any

n , as the rewriting sequence never stops. Such systems are not well-defined, as they do not allow us to evaluate the recurrence symbols on all valuations. We now formalize this point.

WELL-DEFINED GPRS. A GPRS $\mathcal{F}(\mathcal{T}, \mathbf{n})$ is said to be *well-defined* if for all $T \in \mathcal{T}$ and valuation $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$, $T(\sigma(\mathbf{n})) := \sigma(T(\mathbf{n}))$ can be evaluated and every sequence of non-trivial rewrites of $T(\sigma(\mathbf{n}))$ terminates, resulting in a real value.

Apart from being well-defined, we require an additional natural property, called *positivity*. Intuitively speaking, positivity ensures that (i) the final value of any evaluation is positive, and (ii) no instance is ever divided into negatively-many sub-instances, as that makes no sense. Any real-world recurrence system modeling the runtime of a recursive divide-and-conquer algorithm will surely satisfy our positivity requirement.

POSITIVITY. A GPRS $\mathcal{F}(\mathcal{T}, \mathbf{n})$ is *positive* if it has the following two properties:

- (1) For every $T \in \mathcal{T}$ and $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$, every evaluation of $T(\sigma(\mathbf{n}))$ results in a *non-negative* real value;
- (2) Let $\phi := \text{if } (\varphi) \text{ then } T(\mathbf{n}) = R$ be a rewrite rule in $\mathcal{F}$ and $f(\mathbf{n})$ be coefficient of any $\mathcal{M}$ -monomial in R . For all valuations $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$ s.t. $\sigma \models \varphi$, we have $f(\sigma(\mathbf{n})) \geq 0$. Equivalently, $\varphi \implies f(\mathbf{n}) \geq 0$ must always hold.

Example 4. Consider the following GPRS:

$$\mathcal{F} = \left\{ \begin{array}{l} \text{if } (0 \leq n < 5) \text{ then } T(n) = n \\ \text{if } (n \geq 5) \text{ then } T(n) = (n^2 - 25) \cdot T(n/2) + n \end{array} \right\}.$$

It is positive because:

- (1) Every evaluation leads to a non-negative value.
- (2) The implications $0 \leq n < 5 \implies n \geq 0$ and $n \geq 5 \implies n^2 - 25 \geq 0$ are valid.

We do not have any restriction on the polynomial $-n$ present in the second $\mathcal{R}$ -expression, as it is not the coefficient of any $\mathcal{M}$ -monomial.

3 Polynomial Upper-bounds / Templates

We now formally define the concept of polynomial bounds.

POLYNOMIAL UPPER-BOUND. A *polynomial upper-bound* of a GPRS $\mathcal{F}(\mathcal{T}, \mathbf{n})$ is a map $\mathcal{U} : \mathcal{T} \rightarrow \mathbb{R}[\mathbf{n}]$ that associates a polynomial $u_T(\mathbf{n})$ to each recurrence symbol $T \in \mathcal{T}$ such that for every valuation $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$ and every evaluation $\mathcal{E}$ of $T(\sigma(\mathbf{n}))$, we have $u_T(\sigma(\mathbf{n})) \geq \mathcal{E}(T(\sigma(\mathbf{n})))$. To simplify notation, we would use $f(\sigma(\mathbf{n}))$ and $f(\sigma)$ interchangeably.

Example 5. Consider the following GPRS:

@var: n ; **@recurrence:** $T(n)$; **if** $(n \geq 1)$ **then** $T(n) = 2 \cdot T(n/2) + n$
if $(n < 1)$ **then** $T(n) = 1$

A possible polynomial upper-bound for $\mathcal{F}$ is the map $\mathcal{U} : T \mapsto n^2$ since $n^2 \geq \mathcal{E}(T(n))$ for any evaluation of any n .

Our goal in this section is to reduce the problem of synthesizing a polynomial upper-bound to solving a system of polynomial constraints in Nonlinear Real Arithmetic (NRA). This corresponds to deciding a formula in the first-order theory of the reals. We first define a polynomial constraint system $\mathcal{P}$ associated to our GPRS $\mathcal{F}$. Then, in Theorem 1, we prove that if $\mathcal{F}$ is well-defined and positive, then any solution of this polynomial constraint system is a valid upper-bound for $\mathcal{F}$.

TEMPLATES. Let $\mathbb{T}$ be a finite set of variables called *template variables*. A *symbolic polynomial* p in $(\mathbf{n}, \mathbb{T})$ is a polynomial using the variables in $\mathbf{n}$ in which each coefficient is itself a real polynomial in $\mathbb{R}[\mathbb{T}]$. Formally, $p \in \mathbb{R}[\mathbb{T}][\mathbf{n}]$. A *polynomial template* of degree d is a symbolic polynomial p of degree d when seen as a polynomial in $\mathbf{n}$. A template is *canonical* if the coefficient of every $n \in \mathbf{n}$ is a distinct template variable $t_n \in \mathbb{T}$. degree 0 or 1 in the variables $\mathbb{T}$.

Example 6. Let $\mathbf{n} = \{n, m\}$ and $\mathbb{T} = \{t_i \mid 1 \leq i \leq 6\}$. The polynomial $p = t_1 \cdot n^2 + t_2 \cdot n \cdot m + t_3 \cdot m^2 + t_4 \cdot n + t_5 \cdot m + t_6$ is a canonical polynomial template of degree 2 in $(\mathbf{n}, \mathbb{T})$.

SUBSTITUTION. A *symbolic polynomial substitution map* $\mathcal{S} : \mathcal{T} \rightarrow \mathbb{R}[\mathbb{T}][\mathbf{n}]$ is a function that maps each recurrence symbol to a polynomial template of degree d . Given an $\mathcal{R}$ -expression R , we define $\mathcal{S}(R)$ as the expression obtained by substituting every T in R with the polynomial $\mathcal{S}(T)$ and evaluating it.

Example 5 (Continued). Consider the $\mathcal{R}$ -expression $R := 2 \cdot T(n/2) + n$ and let $\mathcal{S} : T \mapsto a \cdot n^2 + b \cdot n + c$ be a symbolic polynomial substitution map. Then, $\mathcal{S}(R) = 2 \cdot (a \cdot n^2/4 + b \cdot n/2 + c) + n$.

POLYNOMIAL CONSTRAINT SYSTEMS. Given a symbolic polynomial substitution map $\mathcal{S}$, the *polynomial constraint system* $\mathcal{P}$ of a GPRS $\mathcal{F}$ with respect to $\mathcal{S}$ is defined as:

$$\mathcal{P} := \{ \text{if } (\varphi) \text{ then } \mathcal{S}(T(\mathbf{n})) \geq \mathcal{S}(R) \mid \\ (\text{if } (\varphi) \text{ then } T(\mathbf{n}) = R) \in \mathcal{F} \}$$

Example 5 (Continued). We have the following polynomial constraint system for $\mathcal{F}$ with respect to $\mathcal{S}$:

$$\mathcal{P} := \left\{ \begin{array}{l} \text{if } (n \geq 1) \text{ then } a \cdot n^2 + b \cdot n + c \geq \frac{a}{2} \cdot n^2 + (b+1) \cdot n + 2 \cdot c \\ \text{if } (n < 1) \text{ then } a \cdot n^2 + b \cdot n + c \geq 1 \end{array} \right\}$$

We observe that $\mathcal{P}$ is simply a constraint system in the first-order theory of the reals.

MODELS. A *model* or *solution* of a polynomial constraint system $\mathcal{P} = \{\phi_1, \dots, \phi_n\}$ is any valuation $\Gamma : \mathbb{T} \rightarrow \mathbb{R}$ for the template variables such that $\Gamma \models \forall \mathbf{n} \bigwedge_{i=1}^n \phi_i$.

Example 5 (Continued). A model for $\mathcal{P}$ is any valuation Γ for $\{a, b, c\}$ satisfying the following constraint:

$$\forall n \quad (n \geq 1 \Rightarrow \Gamma(a) \cdot n^2 + \Gamma(b) \cdot n + \Gamma(c) \geq \frac{\Gamma(a)}{2} \cdot n^2 + (\Gamma(b) + 1) \cdot n + 2 \cdot \Gamma(c))$$

$$\wedge (n < 1 \Rightarrow \Gamma(a) \cdot n^2 + \Gamma(b) \cdot n + \Gamma(c) \geq 1).$$

Suppose the polynomial substitution map $\mathcal{S}$ maps every $T \in \mathcal{T}$ to a canonical template of degree d with fresh template variables. It is easy to verify by definition-chasing that the coefficients in every polynomial upper-bound $\mathcal{U}$ of $\mathcal{F}$ would then provide a solution for the polynomial constraint system $\mathcal{P}$. For a recurrence variable $T \in \mathcal{T}$, let us denote the associated polynomial upper bound $\mathcal{U}(T)$ by $\mathcal{U}_T$. We now prove the converse, i.e. that every model of $\mathcal{P}$ yields a polynomial upper-bound of $\mathcal{F}$, assuming that our GPRS is both well-defined and positive.

Theorem 1. Let $\mathcal{P}$ be the polynomial constraint system of a well-defined and positive GPRS $\mathcal{F}$ with respect to symbolic polynomial substitution map $\mathcal{S} : \mathcal{T} \rightarrow \mathbb{R}[\mathbf{T}][\mathbf{n}]$. If $\Gamma : \mathbb{T} \rightarrow \mathbb{R}$ is a model of $\mathcal{P}$, then $\mathcal{U} := \Gamma(\mathcal{S})$ is a polynomial upper-bound of $\mathcal{F}$.

Proof. We prove that $\mathcal{U}_T(\sigma) \geq \mathcal{E}(T(\sigma))$ for every $T \in \mathcal{T}$, $\sigma : \mathbf{n} \rightarrow \mathbb{R}^+$ and evaluation $\mathcal{E}$ of $T(\sigma)$. Our proof is by induction on the length of the evaluation $\mathcal{E}$.

BASE CASE ($|\mathcal{E}| = 1$). For any $T(\sigma)$, if the evaluation ends in one step, then this must happen by application of a rewrite rule of the form

$\phi := \text{if } (\varphi) \text{ then } T(\mathbf{n}) = R$ such that $\sigma \models \varphi$ and R does not contain any non-terminal, i.e. R is a polynomial in $\mathbf{n}$. In other words, $\phi = \text{if } (\varphi) \text{ then } T(\mathbf{n}) = f(\mathbf{n})$. As the polynomial constraint corresponding to $\text{if } (\varphi) \text{ then } u_T(\mathbf{n}) \geq f(\mathbf{n})$ is valid, we have $u_T(\sigma) \geq f(\sigma) = \mathcal{E}(T(\sigma))$, given that $u_T = \mathcal{U}_T$.

INDUCTION HYPOTHESIS. For every valuation σ and symbol $T \in \mathcal{T}$, if $\mathcal{E}$ is an evaluation of $T(\sigma)$ of length $|\mathcal{E}| \leq k$, then we have $\mathcal{U}_T(\sigma) \geq \mathcal{E}(T(\sigma))$.

INDUCTION STEP. Consider an evaluation $\mathcal{E}$ of $T(\sigma)$ of length $|\mathcal{E}| = k + 1$. Let $\mathcal{E} := R_0 \xrightarrow{\phi_1} R_1 \rightarrow \dots \xrightarrow{\phi_{k+1}} R_{k+1}$. Let $\mathcal{C} := \{V_1(\sigma_1), \dots, V_l(\sigma_l)\}$ be the set of all non-terminals in R_1 where each $V_i \in \mathcal{T}$. Since the evaluation $\mathcal{E}$ ends at a real number R_{k+1} , each of the terms $G_i(\sigma_i(\mathbf{n})) \in \mathcal{C}$ must have the evaluation $\mathcal{E}' = \mathcal{E} \setminus \phi_1$ of length $|\mathcal{E}'| = k$. Thus, by induction hypothesis $\mathcal{U}(V_i)(\sigma_i) \geq \mathcal{E}'(V_i(\sigma_i))$. Now, we just need to apply the positivity properties of GPRS to finish the proof. Let $\phi_1[T(\sigma)] = R_1 := \sigma \left(\sum_{i=1}^t f_i(\mathbf{n}) \cdot M_i + q(\mathbf{n}) \right)$, where each M_i is an $\mathcal{M}$ -monomial. We make two observations:

- (1) Let $M = V_1(\sigma_1) \cdots V_l(\sigma_l)$ be any $\mathcal{M}$ -monomial in R_1 . For each V_i , we have $\mathcal{U}(V_i)(\sigma_i) \geq \mathcal{E}'(V_i(\sigma_i)) \geq 0$ where the last inequality follows from the positivity property. Therefore, it immediately follows by multiplication of such inequalities that $\mathcal{U}(\sigma(M)) \geq \mathcal{E}'(\sigma(M))$.

- (2) The rewrite rule ϕ could have been applied only if $\sigma \models \varphi$. Based on the positivity property, we know that every $f_i(\sigma)$ is non-negative.

Based on these two observations, we have:

$$\begin{aligned}
 \mathcal{U}_T(\sigma) &\geq \mathcal{U}(R_1) && \text{by the constraint system } \mathcal{P} \\
 &= \mathcal{U}\left(\sum_{i=1}^t f_i(\sigma) \cdot \sigma(M_i) + q(\sigma)\right) \\
 &:= \sum_{i=1}^t f_i(\sigma) \cdot \mathcal{U}(\sigma(M_i)) + q(\sigma) \\
 &\geq \sum_{i=1}^t f_i(\sigma) \cdot \mathcal{E}'(\sigma(M_i)) + q(\sigma(\mathbf{n})) \\
 &= \mathcal{E}'(R_1) = \mathcal{E}(T(\sigma))
 \end{aligned}$$

This holds for every evaluation $\mathcal{E}$.

Corollary 1. *If we use a polynomial substitution map $\mathcal{S}$ that maps every $T \in \mathcal{T}$ to a canonical template of degree d with new template variables, then the solutions of $\mathcal{P}$ are in one-to-one correspondence with the upper-bounds of $\mathcal{F}$. In other words, the reduction from synthesizing an upper-bound of $\mathcal{F}$ of degree d to obtaining a solution of $\mathcal{P}$ is both sound and complete.*

Corollary 2. *The problem of synthesis of a polynomial upper-bound of any fixed degree d for an input GPRS $\mathcal{F}$ is decidable.*

Proof. Let $\mathcal{P} = \{\phi_1, \dots, \phi_n\}$. The requirement $(\forall \mathbf{n} \ \bigwedge_{i=1}^n \phi_i)$ is a formula in the first-order theory of the reals, which can be decided by the Tarski-Seidenberg procedure [86,87,52].

4 Our Synthesis Algorithm

In this section, we describe our sound and semi-complete algorithm for automating the synthesis of polynomial upper bounds for a given GPRS. Our algorithm uses theorems from polyhedral and real algebraic geometry. To improve readability, we first provide the overall algorithm and then fill in the mathematical details in the next section. We assume that the input to our algorithm is a GPRS $\mathcal{F}$. Our algorithm consists of the following four steps, which we illustrate by a running example:

STEP 1. INITIALIZING A SYMBOLIC POLYNOMIAL SUBSTITUTION MAP $\mathcal{S}$. The input to our algorithm is a well-defined and positive GPRS $\mathcal{F}(\mathcal{T}, \mathbf{n})$ and a degree bound d . For each recurrence symbol $T \in \mathcal{T}$, we create a *fresh* canonical polynomial template g_T of degree d , where by being *fresh* we mean that no two polynomial templates share any coefficients in $\mathbb{T}$. Let $\mathcal{S}$ be the corresponding symbolic polynomial substitution map.

Example 7. *Suppose the input GPRS $\mathcal{F}$ is the following and the degree bound is $d = 2$.*

@var: n, m ; **@recurrence:** $T(n, m)$;
if $(m \geq 1)$ **then** $T(n, m) = T(n + m, m - 1) + m$
if $(n \geq 0 \wedge m \leq 1)$ **then** $T(n, m) = n$

The symbolic polynomial substitution map $\mathcal{S}$ would consist of the following canonical template of degree 2 for T .

$$\mathcal{S}(T) := a_1 \cdot n^2 + a_2 \cdot n \cdot m + a_3 \cdot m^2 + a_4 \cdot n + a_5 \cdot m + a_6$$

Here, we have $\mathbb{T} = \{a_1, \dots, a_6\}$.

STEP 2. CONSTRUCTING THE POLYNOMIAL CONSTRAINT SYSTEM $\mathcal{P}$.

We compute the polynomial constraint system of $\mathcal{F}$ with respect to $\mathcal{S}$ by substituting each recurrence symbol T with its corresponding polynomial $\mathcal{S}(T)$. Our problem is now reduced to solving a set of constraints of the form

$$\exists \mathbb{T} \quad \forall \mathbf{n} \quad f_1 \geq 0 \wedge \dots \wedge f_r \geq 0 \implies g \geq 0$$

Example 7 (Continued). Our algorithm computes the following two constraints that define $\mathcal{P} := \{\psi_1, \psi_2\}$.

$$\begin{aligned} \text{if } (m \geq 1) \text{ then } & a_1 \cdot n^2 + a_2 \cdot n \cdot m + a_3 \cdot m^2 + a_4 \cdot n + a_5 \cdot m + a_6 \geq \\ & a_1 \cdot (n+m)^2 + a_2 \cdot (n+m) \cdot (m-1) + a_3 \cdot (m-1)^2 + a_4 \cdot (n+m) + \\ & a_5 \cdot (m-1) + a_6 + m \\ \text{if } (n \geq 0 \wedge m \leq 1) \text{ then } & a_1 \cdot n^2 + a_2 \cdot n \cdot m + a_3 \cdot m^2 + a_4 \cdot n \\ & + a_5 \cdot m + a_6 \geq n \end{aligned}$$

STEP 3. ELIMINATING QUANTIFIER ALTERNATION. This is the main step of our algorithm where we apply various tools from polyhedral and real algebraic geometry to reduce our problem to LP/SDP. The techniques in our arsenal are Putinar's Positivstellensatz [83], Handelman's Theorem [51] and the well-known Farkas' Lemma [46]. Note that to apply Handelman's Theorem and Putinar's Postivstellensatz, we have to assume that the set of valuations that satisfy the guard of the constraint forms a compact set⁴. To satisfy this requirement, we can add a constraint of the form $\sum_{n \in \mathbf{n}} n^2 \leq N$ to every guard, where $N \in \mathbb{N}$ is a large number. For example, if all of our parameters are at most 2^{32} , we can use $N = |\mathbf{n}| \cdot 2^{64}$. This will later come in handy for our completeness result, but our approach is sound even without this modification.

Let $\exists \mathbb{T} \forall \mathbf{n} \cdot \varphi \implies g \geq 0$ with $\varphi = f_1 \geq 0 \wedge \dots \wedge f_r \geq 0$ be one of our polynomial constraints with quantifier alternation. Depending upon the degree of the polynomials appearing in this constraint, we apply one of the following cases:

CASE 1: φ AND g ARE LINEAR. In this case we use Farkas' Lemma to reduce this problem to LP. We illustrate this with an example.

⁴ We will define this formally in Section 5.

Example 8 (Application of Farkas' Lemma). *Consider the following constraint:*

$$\text{if } \begin{pmatrix} 2-x+6 \cdot y \geq 0 \\ 1-x-2 \cdot y \geq 0 \end{pmatrix} \text{ then } f(x, y) \geq f(2 \cdot x, y-1) + f(y, x-1) - 1$$

Let us define a template function $f(x, y) = a \cdot x + b \cdot y + c$. The RHS becomes: $f(x, y) - f(2 \cdot x, y-1) - f(y, x-1) + 1 = 2 \cdot b - c + 1 - (a+b) \cdot x - a \cdot y$. By Farkas' lemma, there must exist $\lambda_0, \lambda_1, \lambda_2 \in \mathbb{R}^+ \cup \{0\}$ s.t.

$$\lambda_0 + \lambda_1 \cdot (2 - x + 6 \cdot y) + \lambda_2 \cdot (1 - x - 2 \cdot y) = 2 \cdot b - c + 1 - (a+b) \cdot x - a \cdot y.$$

Intuitively, we are trying to write the RHS of the entailment as a non-negative linear combination of the polynomials in its LHS. After rearranging we get $(\lambda_0 + 2 \cdot \lambda_1 + \lambda_2 - 2 \cdot b + c - 1) - (\lambda_1 + \lambda_2 - a - b) \cdot x + (6 \cdot \lambda_1 - 2 \cdot \lambda_2 + a) \cdot y = 0$. Since this polynomial is identically zero, every one of its coefficients have to be zero, leading to the following LP instance:

$$\begin{aligned} \lambda_0 + 2 \cdot \lambda_1 + \lambda_2 &= 2 \cdot b - c + 1, \\ \lambda_1 + \lambda_2 &= a + b, \\ 6 \cdot \lambda_1 - 2 \cdot \lambda_2 + a &= 0 \\ \lambda_0, \lambda_1, \lambda_2 &\geq 0 \end{aligned}$$

A possible solution would be $\Gamma = \{\lambda_0 \mapsto 0, \lambda_1 \mapsto 1, \lambda_2 \mapsto 4, a \mapsto 2, b \mapsto 3, c \mapsto 1\}$. Substituting the coefficients in the symbolic polynomial gives us the valid bound $f(x, y) := 2 \cdot x + 3 \cdot y + 1$.

CASE 2: φ IS LINEAR. In this case we use Handelman's Theorem (Theorem 3). The intuitive idea is to write the RHS not as a linear combination with non-negative coefficients of the LHS itself, but as a combination of products of the inequalities in the LHS.

Example 9 (Application of Handelman's Theorem). *Consider the following recurrence:*

$$\text{if } (x - y - 1 \geq 0 \wedge y - 1 \geq 0) \text{ then } r(x, y) \geq r(x-1, y) + r(x, y-2) - 9 - x^2 + 3 \cdot y.$$

Let us define the following template function:

$$r(x, y) = a_{20} \cdot x^2 + a_{11} \cdot x \cdot y + a_{02} \cdot y^2 + a_{10} \cdot x + a_{01} \cdot y + a_{00}$$

The consequence (RHS) of the constraint, after some simplification, becomes

$$\begin{aligned} & r(x, y) - r(x-1, y) - r(x, y-2) + 9 + x^2 - 3 \cdot y \\ &= -(a_{20} - 1) \cdot x^2 - a_{11} \cdot x \cdot y - a_{02} \cdot y^2 - \\ & (a_{10} - 2 \cdot a_{11} - 2 \cdot a_{20}) \cdot x - (a_{01} - 4 \cdot a_{02} - a_{11} + 3) \cdot y \\ & - a_{00} + 2 \cdot a_{01} - 4 \cdot a_{02} + a_{10} - a_{20} + 9 \end{aligned}$$

Note that this is a polynomial of degree 2. So, we cannot simply apply a Farkas-like idea and write it as a linear combination of the polynomials on the LHS, which are all of degree 1. Instead, we note that if we assume some polynomials are non-negative, then their multiplication should also be non-negative. Indeed, we can assume the non-negativity of the following polynomials⁵:

$$\{1, x - y - 1, y - 1, (x - y - 1) \cdot (y - 1)\}$$

We now attempt to write the RHS as a linear combination of these polynomials, i.e. we introduce new non-negative variables $\lambda_{00}, \lambda_{10}, \lambda_{01}, \lambda_{11}$ and try to find values for them such that following holds:

$$\begin{aligned} & \lambda_{11} \cdot (x - y - 1) \cdot (y - 1) + \lambda_{10} \cdot (x - y - 1) + \lambda_{01} \cdot (y - 1) + \lambda_{00} \\ & = r(x, y) - r(x - 1, y) - r(x, y - 2) + 9 + x^2 - 3 \end{aligned}$$

Equating all the corresponding coefficients, we get the following linear equations for each monomial:

$$\begin{aligned} x^2 : a_{20} - 1 &= 0 \\ x \cdot y : a_{11} + \lambda_{11} &= 0 \\ y^2 : a_{02} - \lambda_{11} &= 0 \\ x : a_{10} - 2 \cdot a_{11} - 2 \cdot a_{20} + \lambda_{10} - \lambda_{11} &= 0 \\ y : a_{01} - 4 \cdot a_{02} - a_{11} + \lambda_{01} - \lambda_{10} + 3 &= 0 \\ 1 : a_{00} - 2 \cdot a_{01} + 4 \cdot a_{02} - a_{10} + a_{20} + \lambda_{00} - \lambda_{01} - \lambda_{10} + \lambda_{11} - 9 &= 0 \end{aligned}$$

Together with the non-negativity constraints $\lambda_{00}, \lambda_{10}, \lambda_{01}, \lambda_{11} \geq 0$, this forms an LP instance. A solution of our LP is $\Gamma = \{\lambda_{00} \mapsto 5, \lambda_{10} \mapsto 1, \lambda_{01} \mapsto 1, \lambda_{11} \mapsto 1, a_{20} = 1, a_{02} \mapsto 1, a_{11} \mapsto -1, a_{10} \mapsto 0, a_{01} \mapsto 0, a_{00} \mapsto 0\}$. Using this model, we obtain the concrete upper-bound $r(x, y) = x^2 + y^2 - x \cdot y$.

CASE 3: BOTH φ AND g ARE NON-LINEAR. In this case we use Putinar's Positivstellensatz (Theorem 4).

Example 10 (Application of Putinar's Positivstellensatz). Consider the constraint

$$\Phi := \text{if } (x^2 - 10 \geq 0 \wedge x - 1 \geq 0) \text{ then } f(2 \cdot x) \geq 7 \cdot f(x)$$

Let us define a polynomial template: $f(x) = a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$. Again, we cannot simply write the RHS as a linear combination of the polynomials in the guard (LHS) since the RHS is of degree 3 and the guard is of degree 2. Instead, we observe that any sum-of-squares (SOS) polynomial is always guaranteed to be non-negative. Thus, instead of the non-negative coefficients λ_i used in the previous case, we attempt to write the RHS as a combination of the polynomials

⁵ We will later see that these polynomials are a subset of our Handelman monoid.

of the LHS in which the coefficient of each LHS polynomial is itself an SOS polynomial. More formally, we aim to find three SOS polynomials h_0, h_1 , and h_2 such that following holds: $h_0 + h_1 \cdot (x^2 - 10) + h_2 \cdot (x - 1) = f(2 \cdot x) - 7 \cdot f(x)$. Let us now create a template for each h_i in a way that ensures h_i would be an SOS. For simplicity, we set $h_i(x) = (b_{i0} + b_{i1} \cdot x)^2$. The b_{ij} 's are fresh unknowns whose value is yet to be determined. Substituting these templates in the previous equation and simplifying we get:

$$\begin{aligned} & b_{11}^2 \cdot x^4 + (2 \cdot b_{10} \cdot b_{11} + b_{21}^2 - a_3) \cdot x^3 + \\ & (b_{01}^2 + b_{10}^2 - 10 \cdot b_{11}^2 + 2 \cdot b_{20} \cdot b_{21} - b_{21}^2 + 3 \cdot a_2) \cdot x^2 + \\ & (2 \cdot b_{00} \cdot b_{01} - 20 \cdot b_{10} \cdot b_{11} + b_{20}^2 - 2 \cdot b_{20} \cdot b_{21} + 5 \cdot a_1) \cdot x^1 + \\ & (b_{00}^2 - 10 \cdot b_{10}^2 - b_{20}^2 + 6 \cdot a_0) \cdot x^0 = 0. \end{aligned}$$

As before, we obtain an equivalent set of equations by equating each coefficient to zero. On solving, we get the solution as $h_0(x) = 2^2, h_1(x) = 1^2, h_2(x) = x^2$, and finally $f(x) = x^3 - 6$.

In this case, we used a simplified template for each h_i . In general, our algorithm creates a template that captures every possible SOS polynomial. This is a standard technique using Cholesky decomposition and leads to an SDP instance instead of LP. See Section 5 for details.

STEP 4. SOLVING THE LP/SDP. In the last step, we reduced our problem to solving an LP/SDP instance. Note that Cases 1 and 2, which cover the vast majority of real-world recurrences, are reduced to LP and we need SDP only for Case 3. Moreover, both LP and SDP are solvable in PTIME and there are many scalable tools to handle them. In this step, we pass the LP/SDP to an external solver. At this step, the user can also add an objective function to optimize the bound according to their wishes. For example, one can ask the solver to find the bound that minimizes the coefficient of n^2 . We take the solution provided by the solver and plug it back into the template to obtain a polynomial upper-bound for our GPRS. This follows Theorem 1 and concludes our algorithm.

5 Mathematical Toolbox

We now provide a formal overview of the theorems from polyhedral and real algebraic geometry used in Section 4.

SUMS OF SQUARES. A polynomial $h(\mathbf{n})$ is a *sum of squares* (SOS) if it can be written as $h(\mathbf{n}) := p_1^2(\mathbf{n}) + p_2^2(\mathbf{n}) \dots p_r^2(\mathbf{n})$, where each $p_i \in \mathbb{R}[\mathbf{n}]$. We denote the set of all SOS by $\Sigma_2[\mathbf{n}]$.

CLOSURE AND COMPACTNESS. Given a set $X \in \mathbb{R}^n$, the *closure* of X is the set $\overline{X}$ consisting of X and its boundary. A set X is *closed* if $\overline{X} = X$. A set X is *bounded* if there exists $c \in \mathbb{R}$ such that $|x| < c$ for every $x \in X$. A set is *compact* if it is both closed and bounded.

SATISFACTION SET. Given a conjunction φ of polynomial inequalities of the form $f(\mathbf{n}) \geq 0$ or $f(\mathbf{n}) > 0$, we define $SAT(\varphi) := \{\sigma : \mathbf{n} \rightarrow \mathbb{R} \mid \sigma \models \varphi\}$.

Theorem 2 (Farkas' Lemma [46]). *Let $f_1, \dots, f_r$ and g be any linear polynomials over the set of variables $\mathbf{n}$. The implication $f_1 \geq 0 \wedge f_2 \geq 0 \wedge \dots \wedge f_r \geq 0 \implies g \geq 0$ holds for every valuation of $\mathbf{n}$, **if and only if** there exist $\lambda_0, \lambda_1, \dots, \lambda_r$, such that $g \equiv \lambda_0 + \sum_{i=1}^r \lambda_i \cdot f_i$ where all λ_i 's are non-negative reals. Moreover, $f_1 \geq 0 \wedge \dots \wedge f_r \geq 0$ is unsatisfiable **if and only if** $-1 \equiv \lambda_0 + \sum_{i=1}^r \lambda_i f_i$ for some non-negative real λ_i 's.*

HANDELMAN'S THEOREM. We define some terminology useful for our next theorem. Let $F := \{f_1, f_2, \dots, f_r\}$ be a set of polynomials over $\mathbb{V}$. We define the monoid of F as the set:

$$\text{monoid}(F) := \{f_1^{k_1} \cdot f_2^{k_2} \dots f_r^{k_r} \mid k_1, \dots, k_r \in \mathbb{N}\}.$$

In other words, the monoid includes all polynomials that can be obtained as a multiplication of polynomials in F . We also define $\text{monoid}_d(F) := \{f \mid f \in \text{monoid}(F) \wedge \deg(f) \leq d\}$.

Theorem 3 (Handelman's Theorem [51]). *Let $f_1, \dots, f_r$ be any linear polynomials and g be an arbitrary polynomial over $\mathbf{n}$. Additionally, assume that $\psi := f_1 \geq 0 \wedge \dots \wedge f_r \geq 0$ is satisfiable and $\text{SAT}(\psi)$ is compact. If the implication $f_1 \geq 0 \wedge f_2 \geq 0 \wedge \dots \wedge f_r \geq 0 \implies g > 0$ holds for every valuation of $\mathbf{n}$, then there exist λ_i, m_i such that $g \equiv \sum_{i=1}^r \lambda_i \cdot m_i$ where each λ_i is a non-negative real value and each $m_i \in \text{monoid}(\{f_1, \dots, f_r\})$.*

As an example, let $f_1 := (n_1 - 5)$, $f_2 := (n_2 - 10)$, and $g := n_1 \cdot n_2 - 5 \cdot n_2 + 1$. The implication $\phi := f_1 \geq 0 \wedge f_2 \geq 0 \implies g > 0$ holds for all valuations of $\{n_1, n_2\}$. We only use the monoid of degree 2 in this example and have $\text{monoid}_2(\{f_1, f_2\}) = \{1, f_1, f_2, f_1^2, f_2^2, f_1 \cdot f_2\}$. It is easy to verify that $g \equiv 1 + f_1 \cdot f_2 + 10 \cdot f_1$. When relying on Handelman's theorem, our algorithm sets up a template that includes every polynomial in $\text{monoid}_d(F)$ for a user-defined degree d .

Theorem 4 (Putinar's Positivstellensatz [83]). *Let $f_1, \dots, f_r$ and g be polynomials of any degree over $\mathbf{n}$. Suppose $\Psi := f_1 \geq 0 \wedge \dots \wedge f_r \geq 0$ is satisfiable and there is an i such that $\text{SAT}(f_i \geq 0)$ is compact. If $f_1 \geq 0 \wedge f_2 \geq 0 \wedge \dots \wedge f_r \geq 0 \implies g > 0$ holds for all valuations over $\mathbf{n}$, then g can be written as $g \equiv h_0 + \sum_{i=1}^r h_i \cdot f_i$ where each h_i is SOS.*

As an example, let $f_1 := n_1 - 1$, $f_2 := n_2^2 - 5$, and $g := n_1^3 - n_1^2 + n_2^2 + 1$. The boolean implication $\Phi := f_1 \geq 0 \wedge f_2 \geq 0 \implies g > 0$ is valid. We can choose the SOS polynomials $h_0 = 6$, $h_1 = n_1^2$ and $h_2 = 1$, and write $g \equiv h_0 + h_1 \cdot f_1 + h_2 \cdot f_2$. When relying on Putinar's Positivstellensatz in our algorithm, we set up a template for each h_i that can cover all possible SOS of degree d . This is using the following theorems.

Theorem 5 ([14]). *Let $V_{d'}$ be the vector of all monomials of degree at most d' over the variables $\mathbf{n} = \{n_1, \dots, n_k\}$. A polynomial $h \in \mathbb{R}[\mathbf{n}]$ of degree $d := 2 \cdot d'$ is a sum of squares **if and only if** there exists a symmetric positive semi-definite matrix Q such that $h = V_{d'}^T \cdot Q \cdot V_{d'}$.*

Theorem 6 (Cholesky Decomposition, [95]). *A square symmetric matrix Q is positive semi-definite if and only if there exists a lower-triangular matrix L with non-negative diagonal entries such that $Q = L \cdot L^T$.*

To generate a template for an SOS polynomial of degree at most d over the variables $\mathbf{n}$, we first generate a template for the lower-triangular matrix L by creating fresh template variables for each of its non-zero entries. The final template for the SOS polynomial is obtained by setting $h := V_{d'}^T \cdot L \cdot L^T \cdot V_{d'}$. This is a standard technique and the reason why applying Putinar’s Positivstellensatz leads to an SDP instance, instead of an LP instance as in the cases of Farkas and Handelman. See [49, Section 2.7] for examples and [6] for extensions to other combinations of strict/non-strict inequalities. Based on the theorems above, we have the following soundness and semi-completeness result:

Theorem 7 (Soundness and Completeness). *Given a well-defined and positive GPRS $\mathcal{F}(\mathcal{T}, \mathbf{n})$, the algorithm of Section 4 has the following desirable properties:*

- *Soundness: Any solution output by the algorithm is a valid upper-bound. Specifically, any model of the LP/SDP instance leads to a valid polynomial upper-bound for the GPRS.*
- *Semi-completeness: If a polynomial upper-bound $\mathcal{U}$ exists and the degree bounds chosen by the user are large enough, then $\mathcal{U}$ corresponds to one of the models of the LP/SDP instance.*

6 Non-polynomial Bounds

Although our approach is designed for polynomial recurrences and upper-bounds, it is also extensible using standard variable substitution techniques and can thus synthesize exponential bounds or generalized polynomial bounds with rational (non-integral) degrees.

GENERALIZED POLYNOMIALS. Let $\mathbf{n} = \{n_1, \dots, n_k\}$ be a set of non-negative recurrence variables and $q \in \mathbb{N}^+$ a positive integer. We define new variables

$$m_1 = \sqrt[q]{n_1}, \dots, m_k = \sqrt[q]{n_k}.$$

A *generalized q -polynomial* over $\mathbf{n}$ is simply a polynomial over $\{m_1, \dots, m_k\}$. We show the set of generalized q -polynomials over $\mathbf{n}$ by $\mathbb{R}_q[\mathbf{n}] := \mathbb{R}[n_1^{1/q}, \dots, n_k^{1/q}]$. For a generalized q -polynomial p over $\mathbf{n}$, we define the degree of p as its degree over the m_i ’s divided by q .

Example 11. $f(n_1, n_2) = 2 \cdot n_1^{1/5} + 3 \cdot n_2^{2/5}$ is a 5-polynomial over $\{n_1, n_2\}$ with degree $2/5$. Similarly, $g(n_1, n_2) = n_1^{3/6} \cdot n_2^{4/6} + 3 \cdot n_2$ is a 6-polynomial with degree $7/6$.

EXTENDING OUR APPROACH TO GENERALIZED POLYNOMIALS. To find generalized polynomial upper-bounds, it suffices to write our original GPRS in

terms of the new variables, solve it using the algorithm of Section 4, and then rewrite the solution using the original variables.

Example 12. Consider the well-known algorithm of Strassen for multiplying two $n \times n$ matrices [88]. The runtime of this algorithm is classically captured by the following recurrence:

$$\begin{aligned} \text{if } (n < 1) \text{ then } T(n) &= 1 \\ \text{if } (n \geq 1) \text{ then } T(n) &= 7 \cdot T(n/2) + 8 \cdot (n/2)^2 \end{aligned}$$

Introducing $m = \sqrt[6]{n}$, we obtain this equivalent GPRS:

$$\begin{aligned} \text{if } (n < 1 \wedge n = m^6) \text{ then } T(n, m) &= 1 \\ \text{if } (n \geq 1 \wedge n = m^6) \text{ then } T(n, m) &= 7 \cdot T(n/2, m/\sqrt[6]{2}) + 8 \cdot (n/2)^2 \end{aligned}$$

At this point, our algorithm of Section 4 can be applied directly. Nevertheless, it is possible to further simplify the GPRS and eliminate n by defining $G(m) := T(m^6, m)$ to get:

$$\begin{aligned} \text{if } (m < 1) \text{ then } G(m) &= 1 \\ \text{if } (m \geq 1) \text{ then } G(m) &= 7 \cdot G(m/\sqrt[6]{2}) + 8 \cdot (m^6/2)^2 \end{aligned}$$

Applying our algorithm to the latter GPRS leads to the following involved and quite precise bound: $G(m) \leq 0.405 \cdot m^{17} + 6.911 \cdot m^{16} + 1.091 \cdot m^{15} + 9.169 \cdot m^{14} + 0.886 \cdot m^{13} + 5.257 \cdot m^{12} - 0.287 \cdot m^{11} - 0.091 \cdot m^{10} - 2.981 \cdot m^9 - 6.745 \cdot m^8 - 2.226 \cdot m^7 - 5.129 \cdot m^6 - 0.083 \cdot m^5 - 1.543 \cdot m^4 + 1.386 \cdot m^3 + 0.803 \cdot m^2 + 1.787 \cdot m + 1.726$. Writing everything in terms of n we get: $T(n) = G(\sqrt[6]{n}) \leq 0.405 \cdot n^{17/6} + 6.911 \cdot n^{16/6} + 1.091 \cdot n^{15/6} + \dots$. Note that $17/6$ is $2.8\bar{3}$, whereas the best asymptotic bound for Strassen is $O(n^{\log_2 7})$ and $\log_2 7 \approx 2.81$. Thus, our algorithm is able to find an almost optimal bound. Due to our semi-completeness, by increasing q , we can get as close to $\log_2 7$ as desired.

EXPONENTIAL BOUNDS. Applying the standard technique of log-substitution, our approach is able to obtain exponential upper-bounds as well.

Example 13. Consider this GPRS:

$$\begin{aligned} \text{if } (n \leq 1) \text{ then } F(n) &= 1 \\ \text{if } (n \geq 2) \text{ then } F(n) &= 2 \cdot F(n-1) + 3 \cdot F(n-2) \end{aligned}$$

Define $m := 3^n$ so that we have $n = \log_3 m$. Also define $G(m) := F(\log_3(m))$, leading to this equivalent GPRS:

$$\begin{aligned} \text{if } (1 \leq m \leq 3) \text{ then } G(m) &= 1 \\ \text{if } (m \geq 9) \text{ then } G(m) &= 2 \cdot G(m/3) + 3 \cdot G(m/9) \end{aligned}$$

Now that every expression in the GPRS is polynomial, we apply our algorithm, which obtains the upper-bound $G(m) \leq m$. Substituting back, we obtain $F(n) \leq 3^n$. This method can be applied to systems of linear recurrences of arbitrary order.

7 Experimental Results

Our main theoretical contribution in this work is the completeness of our algorithm. Such completeness results usually come at the cost of sacrificing practicality. For example, decision procedures for the existential theory of the reals are notoriously unscalable and not applicable to any real-world or even toy use-case. In contrast, our algorithm reduces the problem to LP/SDP, for which there are plenty of efficient solvers. Thus, we now provide experimental results demonstrating that, in addition to its completeness, our approach is practical and efficiently solves a variety of recurrence systems taken from three categories in the literature. We implemented our algorithm in C++ and used the Mosek Optimization Suite [79] to solve LP/SDP instances. Our tool is free and open-source and can be accessed at [81] (<https://doi.org/10.5281/zenodo.14836308>). All results were obtained on an AMD Ryzen 5 4600H Machine (3 GHz, 12 cores) running Ubuntu 20.04 LTS with 16 GB of RAM.

A. INVARIANT GENERATION IN PROBABILISTIC PROGRAMS. Our first category of benchmark GPRSs come from invariant generation. The works [10,78] provide an algorithm and tool, called Polar, to generate moment-based invariants for a wide family of loops in probabilistic programs, called prob-solvable loops. The algorithm first extracts a recurrence system which connects the moments of program variables and then attempts to solve it.

```

y = 0; x = 0
while true:
  y = y + 1
  x = x + y if uniform(0, 1) > 0.5 else x

```

As the simplest example, consider the program above, where `uniform(0, 1)` denotes a sample taken from the uniform distribution on $[0, 1]$. This loop computes part of the summation $\sum_{y=1}^{\infty} y$ where each term is taken with probability $1/2$. Let $Ex(n)$ and $Ey(n)$ denote the expected values of the variables after n iterations. The work [10] obtains the following recurrence system:

```

if (n = 0) then  $Ex(n) = 0$ 
if (n = 0) then  $Ey(n) = 0$ 
if (n ≥ 1) then  $Ex(n + 1) = Ex(n) + Ey(n - 1)/2 + 1/2$ 
if (n ≥ 1) then  $Ey(n + 1) = Ey(n) + 1$ 

```

Our approach can be directly applied to this GPRS and provides the following solution: $Ex(n) = 0.25 \cdot n^2 + 0.25 \cdot n$, and $Ey(n) = n$.

Table 1 provides a list of probabilistic programs from [10] that can be encoded in our GPRS framework. Our approach could successfully and efficiently solve these recurrence systems and compute the moments. Table 1 shows that (i) our approach is able to handle systems of recurrences which are beyond the scope of MT and AB, and (ii) in addition to completeness, we preserve practicality and solve each recurrence system within a few seconds, with most cases taking less than a second. In comparison, Polar is efficient but lacks completeness.

Table 1: Results on GPRSs from Prob-solvable loops. degree in templates.

Program		D	Our time (s)	Polar (s)	Moments
sum_rnd_series	1	0.058	0.426	$\mathbb{E}x, \mathbb{E}y$	
	2	0.187	0.733	$\mathbb{E}x^2, \mathbb{E}xy, \mathbb{E}y^2$	
random_walk_1d_intvalued	0	0.032	0.413	$\mathbb{E}x$	
	1	0.048	0.476	$\mathbb{E}x^2$	
random_walk_1d_realvalued	0	0.036	0.416	$\mathbb{E}x$	
	1	0.049	0.461	$\mathbb{E}x^2$	
product_dep_var	1	0.303	0.488	$\mathbb{E}p, \mathbb{E}xy$	
	2	0.952	0.787	$\mathbb{E}p^2, \mathbb{E}x^2y^2$	
random_walk_2d	0	0.090	0.202	$\mathbb{E}x, \mathbb{E}y$	
	0	0.091	0.388	$\mathbb{E}x^2, \mathbb{E}y^2$	
binomial	1	0.145	0.409	$\mathbb{E}x$	
	2	0.447	0.498	$\mathbb{E}x^2$	
stuttering_p	1	1.363	0.467	$\mathbb{E}s, \mathbb{E}x, \mathbb{E}y$	
	2	4.483	0.885	$\mathbb{E}s^2, \mathbb{E}x^2, \mathbb{E}xy, \mathbb{E}y^2$	
square	1	0.216	0.466	$\mathbb{E}y, \mathbb{E}x^2$	
	2	0.550	0.751	$\mathbb{E}y^2, \mathbb{E}x^4$	

Conversely, it can handle non-polynomial recurrences which are not supported in our GPRS framework.

B. RESOURCE USAGE OF FUNCTIONAL PROGRAMS. RaML [54,55] is a framework based on type theory to analyze and bound the resource consumption of functional programs written in OCaml. The resource in question can be time or space or arbitrarily defined by assigning costs to certain operations in the program. This was later extended in PUBS [4,5]. As before, these approaches first find a system of recurrences and then solve it to find a bound, e.g. this GPRS models a program performing the sieve of Eratosthenes [55]:

```

if (A = 0) then eratos(A) = 1
if (B = 0) then filter(X, B) = 1
if (A ≥ 1) then eratos(A) = 1 + eratos(A - 1) + filter(X, A - 1)
if (B ≥ 1) then filter(X, B) = 1 + filter(X, B - 1)

```

This example cannot be handled by either MT or AB, since it has two functions referring to each other and a function with two variables. Our approach obtains the bounds $\text{eratos}(n) \leq 0.5 \cdot n^2 + 1.5 \cdot n + 1$ and $\text{filter}(x, m) \leq m + 1$.

Table 2 provides a list of functional programs from [54,55] whose resource usage can be modeled by a polynomial recurrence system. It also reports the resource usage bounds obtained by our approach and the previous tools, especially CHORA [15], which is the closest related work. Table 2 shows that (i) our approach is practical and successfully handles non-trivial systems of recurrences, (ii) the bounds obtained by our approach on these programs are either tighter than the previous methods, the same, or incomparable. Recall that our approach is able to find the best polynomial bound w.r.t. any user-defined objective function. Thus, it is able to synthesize the same polynomial bounds as previous tools,

too. In Table 2, we are not reporting the runtimes since the results were obtained using previous methods’ web interfaces and not on our machine. PUBS is proprietary and no source code or executable is available for it. Thus, we do not claim to be faster than them, but rather to be the first approach for polynomial recurrences with both practicality and completeness guarantees.

C. CLASSICAL ALGORITHMS. We consider four classical algorithms for which the best known runtime bound is not polynomial. This includes Strassen’s matrix multiplication algorithm, Karatsuba’s multiplication, fast exponentiation by squaring, and merge sort. We used the following recurrences:

$$\begin{aligned}\mathcal{F}_{\text{Stra.}} &= \begin{cases} \text{if } (0 \leq n < 1) \text{ then } T(n) = 1 \\ \text{if } (n \geq 1) \text{ then } T(n) = 7 \cdot T(n/2) + 2 \cdot n^2 \end{cases} \\ \mathcal{F}_{\text{Merg.}} &= \begin{cases} \text{if } (0 \leq n < 1) \text{ then } T(n) = 1 \\ \text{if } (n \geq 1) \text{ then } T(n) = 2 \cdot T(n/2) + n + 1 \end{cases} \\ \mathcal{F}_{\text{Kara.}} &= \begin{cases} \text{if } (0 \leq n < 1) \text{ then } T(n) = 1 \\ \text{if } (n \geq 1) \text{ then } T(n) = 3 \cdot T(n/2) + n + 1 \end{cases} \\ \mathcal{F}_{\text{Exp.}} &= \begin{cases} \text{if } (0 \leq n < 1) \text{ then } T(n) = 1 \\ \text{if } (n \geq 1) \text{ then } T(n) = T(n/2) + 4 \end{cases}\end{aligned}$$

Table 3 shows our experimental results in comparison with CHORA [15]. Although our approach cannot find the optimal answer since it is not polynomial, using the trick of Section 6 works well in practice and obtains close-to-optimal results. We generally observe that CHORA is faster. This is due to the high degree of polynomials needed in templates to model non-integer degrees. Conversely, our completeness pays off in practice and our approach is able to find a tighter bound for fast exponentiation.

D. ABLATION STUDY. Finally, we performed an ablation study to measure the practical impact of Step 3 of our algorithm, i.e. the application of Farkas, Handelman and Putinar theorems. To this end, we passed the constraints generated by the end of Step 2, which are formulas in the first-order theory of the reals with an $\exists\forall$ quantifier alternation, directly to state-of-the-art SMT solvers, including Z3 [80], cvc4 [9], cvc5 [8] and Vampire [73]. The latter two are the winners of the non-linear real arithmetic competition track of SVCOMP [13]. With a time limit of 12 hours per solver per instance, none of the solvers were able to handle our benchmarks. Thus, the dedicated quantifier elimination methods are necessary not only theoretically but also in practice.

In summary, in addition to providing completeness guarantees for systems of polynomial recurrences, our approach remains practical. To our knowledge, polynomial recurrence systems are one of the most expressive families of recurrences to date to be solved by a complete method and are not subsumed by any of the previous approaches. We also showed that our approach successfully synthesizes bounds for benchmarks that were beyond the scope of previous complete methods, i.e. MT and AB (Tables 1 and 2) and that it can find close-to-optimal bounds even in cases where the optimal solution is not a polynomial (Table 3). Additionally, due to its completeness, it often finds tighter bounds than previous methods and can handle instances that were not solvable by CHORA.

Table 2: Results on GPRs of RaML for Resource Usage of Functional Programs.

Program	D Time (s)	Function	Our Bound	PUBS	RaML	CHORA
eratohenes	1 0.14	eratos(n) filter(x, m)	$0.5 \cdot n^2 + 1.5 \cdot n + 1.0$ $1.0 \cdot m + 1.0$	$1.0 \cdot n^2 + 1.0 \cdot n + 1.0$ $1.0 \cdot m + 1.0$	$0.5 \cdot n^2 + 1.5 \cdot n + 1.0$ $1.0 \cdot m + 1.0$	$0.5 \cdot n^2 + 1.5 \cdot n + 1.0$ $1.0 \cdot m + 1.0$
		add(n, m) subp(n, m, x) compare(n, m) leq(n, m)	$1.88 \cdot n + 0.12 \cdot m + 4.94$ $2.04 \cdot n + 0.01 \cdot m + 1.13$ $1.02 \cdot n + 0.01 \cdot m + 1.13$ $1.1 \cdot n + 0.08 \cdot m + 1.19$	$2.0 \cdot m + 2.0$ $2.0 \cdot m + 1.0$ $1.0 \cdot m + 1.0$ $1.0 \cdot m + 1.0$	$2.0 \cdot n + 2.0$ $2.0 \cdot m + 1.0$ $1.0 \cdot m + 1.0$ $1.0 \cdot m + 2.0$	$2.0 \cdot m + 2.0$ $2.0 \cdot m + 1.0$ $1.0 \cdot n + 1.0$ $1.0 \cdot n + 1.0$
bitvectors	2 43.86	mult(n, m) mult3(n, m, x)	$0.778 \cdot n \cdot m + 25.878 \cdot n + 1.0$ $1.556 \cdot n \cdot m + 51.756 \cdot n + 3.0$	$2.0 \cdot n^2 + 3.0 \cdot n + 1.0$ $4.0 \cdot n^2 + 6.0 \cdot n + 3.0$	$2.0 \cdot m \cdot n + 3.0 \cdot n + 1.0$ $3 + 2 \cdot m \cdot n + 6 \cdot n + 2.0 \cdot n \cdot x$	$1.0 \cdot n^2 + 2.0 \cdot n + 1.0$ $2.0 \cdot n^2 + 4.0 \cdot n + 3.0$
		eq(n, m) remove(n, m, x) nub(n, m)	$1.0 \cdot n + 1.0$ $41.52 \cdot m + 1.0$ $4.29 \cdot n^3 + 4.28 \cdot n^2 +$	$1.0 \cdot m + 1$ $1.0 \cdot m \cdot x + 2.0 \cdot m + 1$ $0.5 \cdot m \cdot n^2 + 1.0 \cdot n^2 +$	$1.0 \cdot n + 1.0$ $1.0 \cdot m \cdot x + 2.0 \cdot m + 1.0$ $0.5 \cdot n^2 \cdot m + 1.0 \cdot n^2 -$	$1.0 \cdot m + 1$ $1.0 \cdot n + 1.0$ Failed
duplicates	2 17.27		$0.06 \cdot n \cdot m^2 + 1.62 \cdot n + 1.0$	$0.5 \cdot m \cdot n + 3.0 \cdot n + 1.0$	$0.5 \cdot n \cdot m + 1.0 \cdot n + 1.0$	
lcs	1 0.48	lcs(n, m)	$4.0 \cdot n \cdot m + 2.0 \cdot n +$ $1.0 \cdot m + 3.0$	$4.0 \cdot n \cdot m + 2.0 \cdot n +$ $1.0 \cdot m + 3.0$	$4.0 \cdot n \cdot m + 2.0 \cdot n +$ $1.0 \cdot m + 3.0$	$4.0 \cdot n \cdot m + 2.0 \cdot n + m + 3.0$
minsort	1 0.13	minSort(n, m) findMin(n)	$0.5 \cdot n^2 + 2.5 \cdot n + 2.0$ $1.0 \cdot n + 1.0$	$0.5 \cdot n^2 + 2.5 \cdot n + 2$ $1.0 \cdot n + 1.0$	$0.5 \cdot n^2 + 2.5 \cdot n + 2$ $1.0 \cdot n + 1.0$	$0.5 \cdot n^2 + 2.5 \cdot n + 2$ $1.0 \cdot n + 1.0$
rational	1 0.28	group3(n) zip3(n, m, x)	$0.33333 \cdot n$ $1.0 \cdot n$	$0.33333 \cdot n$ $1.0 \cdot n$	$0.33333 \cdot n$ $1.0 \cdot n$	$1.0 \cdot n - 3$ $\max(n, m, x)$
Potential						

Table 3: Experimental Results on GPRs of Classical Algorithms. D is the maximum degree in templates. PUBS time measurements were not conducted as the tool is available only through a web interface.

Algorithm	D	Our Time (s)	Our Bound	CHORA's Time (s)	CHORA's Bound	RaML's Time (s)	RaML's Bound	PUBS Bound	Optimal
Strassen	8	274.74	$0.405 \cdot n^{2.833} + \dots$	–	Failed	–	Failed	$0.16 \cdot n^{\log_2 7} O(n^{2.807\dots}) = O(n^{\log_2 7})$	
Merge Sort	5	20.90	$0.003 \cdot n^{1.1} + \dots$	2.15	$2.0 \cdot n \cdot \log n + 1.0$	0.11	n^2	n^2	$O(n \cdot \log n)$
Karatsuba	4	6.51	$7.937 \cdot n^{1.6} + \dots$	2.09	$10.5 \cdot n^{1.585\dots}$	0.14	$1.5 \cdot n^2 - 0.5 \cdot n$	$n^{\log_2 3}$	$O(n^{1.585\dots}) = O(n^{\log_2 3})$
Exponentiation	8	322.36	$42.455 \cdot n^{0.143} + \dots$	0.47	$4.0 \cdot n$	0.03	$0.5 \cdot n$	$\log n$	$O(\log n)$

Table 4: Ablation results on the benchmarks. All times are in seconds. The time limit for each solver and each instance 12 hours.

Program	Our	Z3	cvc4	cvc5	Vampire
sum_rnd_	0.06	0.12	0.12	0.22	TL
series	0.19	TL	25.25	TL	TL
random_walk_	0.03	0.18	0.07	0.08	TL
1d_intvalued	0.05	0.03	0.06	TL	TL
random_walk_	0.04	0.14	0.07	0.13	TL
1d_realvalued	0.05	0.06	0.11	TL	TL
product_	0.30	0.39	2.08	TL	TL
dep_var	0.95	TL	TL	TL	TL
random_walk_	0.09	1.45	12.7	TL	TL
2d	0.09	TL	TL	TL	TL
binomial	0.14	0.23	0.23	TL	TL
	0.45	TL	TL	TL	TL
stuttering_p	1.36	TL	TL	TL	TL
	4.48	TL	TL	TL	TL
square	0.22	TL	0.57	TL	TL
	0.55	TL	TL	TL	TL
eratosthenes	0.14	0.16	TL	TL	TL
bitvectors	43.86	TL	TL	TL	TL
duplicates	17.27	TL	TL	TL	TL
lcs	0.48	TL	TL	TL	TL
minsort	0.13	0.19	0.13	0.69	TL
rationalPotential	0.28	0.17	TL	TL	TL
Strassen	274.74	TL	TL	TL	TL
Merge Sort	20.90	TL	TL	TL	TL
Karatsuba	6.51	TL	TL	TL	TL
Exponentiation	322.36	TL	TL	TL	TL

8 Conclusion

In this work, we considered Generalized Polynomial Recurrence Systems (GPRS), a wide family of recurrences that allows (i) a system of interdependent recurrence relations, (ii) an arbitrary number of variables, and (iii) conditional recurrence rules. We provided an automated template-based algorithm for finding polynomial solutions, i.e. upper-bounds, for such recurrences. Our algorithm is sound and semi-complete and guaranteed to find the tightest possible polynomial bounds with respect to any user-defined objective function. Moreover, for any fixed degree of polynomial templates, its complexity is PTIME.

ACKNOWLEDGMENTS AND NOTES. We are grateful to the anonymous ESOP reviewers for their detailed comments, which helped improve this work significantly. The research was partially supported by an Ethereum Foundation Research Grant. Authors are ordered alphabetically.

References

1. Akra, M.A., Bazzi, L.: On the solution of linear recurrence equations. *Comput. Optim. Appl.* **10**(2), 195–210 (1998)
2. Akshay, S., Chakraborty, S., Goharshady, A.K., Govind, R., Motwani, H.J., Varanasi, S.T.: Automated synthesis of decision lists for polynomial specifications over integers. In: *LPAR*. pp. 484–502 (2024)
3. Akshay, S., Chakraborty, S., Goharshady, A.K., Govind, R., Motwani, H.J., Varanasi, S.T.: Practical approximate quantifier elimination for non-linear real arithmetic. In: *FM*. pp. 111–130 (2024)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: *SAS*. pp. 221–237 (2008)
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: PUBS: a practical upper bounds solver. <https://costa.fdi.ucm.es/~costa/pubs/examples.php> (2008)
6. Asadi, A., Chatterjee, K., Fu, H., Goharshady, A.K., Mahdavi, M.: Polynomial reachability witnesses via stellensätze. In: *PLDI*. pp. 772–787 (2021)
7. Atkey, R.: Amortised resource analysis with separation logic. *Log. Methods Comput. Sci.* **7**(2) (2011)
8. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: *TACAS*. pp. 415–442 (2022)
9. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: *CAV*. pp. 171–177 (2011)
10. Bartocci, E., Kovács, L., Stankovic, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: *ATVA*. pp. 255–276 (2019). https://doi.org/10.1007/978-3-030-31784-3_15
11. Basu, S., Pollack, R., Roy, M.F.: Existential theory of the reals. *Algorithms in Real Algebraic Geometry* pp. 505–532 (2006)
12. Bentley, J.L., Haken, D., Saxe, J.B.: A general method for solving divide-and-conquer recurrences. *SIGACT News* **12**(3), 36–44 (1980)
13. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: *TACAS*. pp. 495–522 (2023)
14. Blekherman, G., Parrilo, P.A., Thomas, R.R.: Semidefinite optimization and convex algebraic geometry (2012)
15. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.W.: Templates and recurrences: better together. In: *PLDI*. pp. 688–702 (2020)
16. Cai, Z., Farokhnia, S., Goharshady, A.K., Hitarth, S.: Asparagus: Automated synthesis of parametric gas upper-bounds for smart contracts. *Proc. ACM Program. Lang.* **7**(OOPSLA2), 882–911 (2023)
17. Campbell, B.: Amortised memory analysis using the depth of data structures. In: *ESOP*. vol. 5502, pp. 190–204 (2009)
18. Canny, J.: Some algebraic and geometric computations in PSPACE. In: *STOC*. pp. 460–467 (1988)
19. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for C programs. In: *PLDI*. pp. 270–281 (2014)
20. Carbonneaux, Q., Hoffmann, J., Reps, T.W., Shao, Z.: Automated resource analysis with Coq proof objects. In: *CAV*. pp. 64–85 (2017)
21. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: *PLDI*. pp. 467–478 (2015)

22. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reason.* **62**(3), 331–365 (2019)
23. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through positivstellensatz’s. In: *CAV*. pp. 3–22 (2016)
24. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. In: *CAV*. vol. 10427, pp. 41–63 (2017)
25. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. *ACM Trans. Program. Lang. Syst.* **41**(4), 20:1–20:52 (2019)
26. Chatterjee, K., Fu, H., Goharshady, A.K., Goharshady, E.K.: Polynomial invariant generation for non-deterministic recursive programs. In: *PLDI*. pp. 672–687 (2020)
27. Chatterjee, K., Fu, H., Goharshady, A.K., Okati, N.: Computational approaches for stochastic shortest path on succinct MDPs. In: *IJCAI*. pp. 4700–4707 (2018)
28. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: *POPL*. pp. 327–342 (2016)
29. Chatterjee, K., Goharshady, A.K., Goharshady, E.K., Karrabi, M., Zikelic, D.: Sound and complete witnesses for template-based verification of LTL properties on polynomial programs. In: *FM*. pp. 600–619 (2024)
30. Chatterjee, K., Goharshady, A.K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.: Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. *ACM Trans. Program. Lang. Syst.* **41**(4), 23:1–23:46 (2019)
31. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In: *POPL*. pp. 733–747 (2016)
32. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Zikelic, D.: Sound and complete certificates for quantitative termination analysis of probabilistic programs. In: *CAV*. pp. 55–78 (2022)
33. Chatterjee, K., Goharshady, A.K., Meggendorfer, T., Zikelic, D.: Quantitative bounds on resource usage of probabilistic programs. *Proc. ACM Program. Lang.* **8**(OOPSLA1), 362–391 (2024)
34. Chatterjee, K., Ibsen-Jensen, R., Goharshady, A.K., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. *ACM Trans. Program. Lang. Syst.* **40**(3), 9:1–9:43 (2018)
35. Chaudhuri, S., Dubhashi, D.P.: Probabilistic recurrence relations revisited. *Theor. Comput. Sci.* **181**(1), 45–56 (1997)
36. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: *CAV*. pp. 420–432 (2003)
37. Conrado, G.K., Goharshady, A.K., Kochev, K., Tsai, Y.C., Zaher, A.K.: Exploiting the sparseness of control-flow and call graphs for efficient and on-demand algebraic program analysis. *Proc. ACM Program. Lang.* **7**(OOPSLA2), 1993–2022 (2023)
38. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to algorithms* (1990)
39. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: *Parameterized algorithms* (2015)
40. Cyphert, J., Kincaid, Z.: Solvable polynomial ideals: The ideal reflection for program analysis. *Proc. ACM Program. Lang.* **8**(POPL), 724–752 (2024)
41. Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: *CSF*. pp. 1–16 (2021)

42. Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session types. In: LICS. pp. 305–314 (2018)
43. Das, A., Qadeer, S.: Exact and linear-time gas-cost analysis. In: SAS. vol. 12389, pp. 333–356 (2020)
44. Downey, R.G., Fellows, M.R.: Parameterized complexity (2012)
45. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. *J. ACM* **57**(6), 33:1–33:47 (2010)
46. Farkas, J.: Theorie der einfachen ungleichungen (in German). *Journal für die reine und angewandte Mathematik* **190**(124), 1–27 (1902)
47. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD. pp. 57–64 (2015)
48. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. *J. ACM* **56**(5), 25:1–25:32 (2009)
49. Goharshady, A.: Parameterized and Algebro-geometric Advances in Static Program Analysis. Ph.D. thesis, Institute of Science and Technology Austria (Nov 2020), <https://hal.science/tel-03153209>
50. Goharshady, A.K., Hitarth, S., Mohammadi, F., Motwani, H.J.: Algebro-geometric algorithms for template-based synthesis of polynomial programs. In: OOPSLA. pp. 727–756 (2023)
51. Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific Journal of Mathematics* **132**(1), 35–62 (1988)
52. Heintz, J., Roy, M.F., Solernó, P.: Sur la complexité du principe de Tarski-Seidenberg (in French). *Bulletin de la Société mathématique de France* **118**(1), 101–126 (1990)
53. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* **34**(3), 14:1–14:62 (2012)
54. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: CAV. pp. 781–786 (2012). https://doi.org/10.1007/978-3-642-31424-7_64
55. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. <https://www.raml.co/> (2012)
56. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In: APLAS. pp. 172–187 (2010)
57. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: ESOP. pp. 287–306 (2010)
58. Hoffmann, J., Jost, S.: Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* **32**(6), 729–759 (2022)
59. Hofmann, M.: A type system for bounded space and functional in-place update. *Nord. J. Comput.* **7**(4), 258–289 (2000)
60. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL. pp. 185–197 (2003)
61. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: ESOP. pp. 22–37 (2006)
62. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: CSL. pp. 317–331 (2009)
63. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.* **3**(OOPSLA), 129:1–129:29 (2019)
64. Humenberger, A., Jaroschek, M., Kovács, L.: Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In: ISSAC. pp. 221–228 (2017)

65. Ishimwe, D., Nguyen, K., Nguyen, T.: Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. *Proc. ACM Program. Lang.* **5**(OOPSLA), 1–23 (2021)
66. Jost, S., Vasconcelos, P.B., Florido, M., Hammond, K.: Type-based cost analysis for lazy functional languages. *J. Autom. Reason.* **59**(1), 87–120 (2017)
67. Kahn, D.M., Hoffmann, J.: Exponential automatic amortized resource analysis. In: *FoSSaCS*. pp. 359–380 (2020)
68. Karp, R.M.: Probabilistic recurrence relations. In: *STOC*. pp. 190–197. ACM (1991)
69. Karp, R.M.: Probabilistic recurrence relations. *J. ACM* **41**(6), 1136–1150 (1994)
70. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: *PLDI*. pp. 248–262 (2017)
71. Kincaid, Z., Reps, T.W., Cyphert, J.: Algebraic program analysis. In: *CAV*. pp. 46–83 (2021)
72. Kovács, L.: Reasoning algebraically about p-solvable loops. In: *TACAS*. pp. 249–264 (2008)
73. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: *CAV*. pp. 1–35 (2013)
74. Kovács, L., Jebelean, T.: Automated generation of loop invariants by recurrence solving in theoremata (01 2004)
75. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. *Artif. Intell.* **235**, 1–25 (2016)
76. Liu, H., Fu, H., Yu, Z., Song, J., Li, G.: Scalable linear invariant generation with Farkas’ lemma. In: *OOPSLA*. pp. 204–232 (2022)
77. Macintyre, A., Wilkie, A.J., Odifreddi, P.: On the decidability of the real exponential field. *Kreisel’s Mathematics* **115**, 451 (1996)
78. Moosbrugger, M., Amrollahi, D., Müllner, J., Kenison, G., Kofnov, A.: Polar github repository. <https://github.com/probing-lab/polar> (2023)
79. MOSEK ApS: Mosek optimization suite (2023), <https://docs.mosek.com/10.0/intro.pdf>
80. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: *TACAS*. pp. 337–340 (2008)
81. Novozhilov, S., Hitarth, S., Kafshdar Goharshady, A.: polyrecur: A C++ tool based on SDP solvers for solving generalized polynomial recurrence systems (GPRS) (2025). <https://doi.org/10.5281/zenodo.14836308>
82. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: *VMCAI*. vol. 2937, pp. 239–251 (2004)
83. Putinar, M.: Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal* **42**(3), 969–984 (1993)
84. Reps, T.W., Turetsky, E., Prabhu, P.: Newtonian program analysis via tensor product. In: *POPL*. pp. 663–677 (2016)
85. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: *POPL*. pp. 318–329 (2004)
86. Schoutens, H.: Muchnik’s proof of Tarski-Seidenberg. Ohio State University (2001)
87. Semenov, A.: Decision procedures for logical theories (in Russian). *Cybernetics and computer technology* **2**, 134–146 (1986)
88. Strassen, V.: Gaussian elimination is not optimal. *Numerische Mathematik* **13**(4), 354–356 (1969)
89. Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Automated tail bound analysis for probabilistic recurrence relations. In: *CAV*. pp. 16–39 (2023)

90. Tassarotti, J., Harper, R.: Verified tail bounds for randomized programs. In: ITP. pp. 560–578 (2018)
91. Wang, C., Lin, F.: Solving conditional linear recurrences for program verification: The periodic case. In: OOPSLA. pp. 28–55 (2023)
92. Wang, D., Kahn, D.M., Hoffmann, J.: Raising expectations: automating expected cost analysis with types. *Proc. ACM Program. Lang.* **4**(ICFP), 110:1–110:31 (2020)
93. Wang, J., Sun, Y., Fu, H., Chatterjee, K., Goharshady, A.K.: Quantitative analysis of assertion violations in probabilistic programs. In: PLDI. pp. 1171–1186 (2021)
94. Wang, P., Fu, H., Goharshady, A.K., Chatterjee, K., Qin, X., Shi, W.: Cost analysis of nondeterministic probabilistic programs. In: PLDI. pp. 204–220 (2019)
95. Watkins, D.S.: *Fundamentals of matrix computations* (2004)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





CUTECat: Concolic Execution for Computational Law

Pierre Goutagny¹(✉) , Aymeric Fromherz² , and Raphaël Monat¹ 

¹ Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, 59000 Lille, France
pierre.goutagny@inria.fr, raphael.monat@inria.fr

² Inria Paris, Paris, France
aymeric.fromherz@inria.fr

Abstract. Many legal computations, including the amount of tax owed by a citizen, whether they are eligible to social benefits, or the wages due to civil state servants, are specified by *computational laws*. Their application, however, is performed by expert computer programs intended to faithfully transcribe the law into computer code. Bugs in these programs can lead to dramatic societal impact, e.g., paying employees incorrect amounts, or not awarding benefits to families in need.

To address this issue, we consider concolic unit testing, a combination of concrete execution with SMT-based symbolic execution, and propose CUTECat, a concolic execution tool targeting implementations of computational laws. Such laws typically follow a pattern where a base case is later refined by many exceptions in following law articles, a pattern that can be formally modeled using *default logic*. We show how to handle default logic inside a concolic execution tool, and implement our approach in the context of Catala, a recent domain-specific language tailored to implement computational laws. We evaluate CUTECat on several programs, including the Catala implementation of the French housing benefits and Section 132 of the US tax code. We show that CUTECat can successfully generate hundreds of thousands of testcases covering all branches of these bodies of law. Through several heuristics, we improve CUTECat’s scalability and usability, making the testcases understandable by lawyers and programmers alike. We believe CUTECat paves the way for the use of formal methods during legislative processes.

1 Introduction

Since at least the Sumerian empire and the code of Ur-Nammu [57], human societies have been governed by laws. From constitutional law to environmental law, including criminal law, immigration law or intellectual property law, laws are applied in a wide range of contexts, representative of the diverse facets of modern societies. Laws are typically stated in natural language, e.g., English, and therefore require human interpretation to determine when and how they must apply: to do so, a criminal trial might rely on a grand jury to determine the guilt of a defendant, while companies typically hire highly specialized lawyers to ensure that a merger is performed according to corporate law.

Other laws, however, focus on defining well-specified computations, for instance, the amount of taxes that a household owes, the wages of civil state servants, or whether a given family is eligible to social benefits depending on their situation. As they must be applied to a large number of citizens, such laws, commonly known as *computational laws*, are typically implemented in expert computer systems, which are able to automate the computation for any input.

Unfortunately, as for any computer program, legal expert systems are not immune to bugs, which can have tremendous consequences, either for the states or their citizens. Case in point, more than half of all Canadian civil servants suffered pay issues from the Phoenix automated payroll system, resulting in years of fixing incurred financial issues for civil servants, and, for the Canadian government “\$2.2 billion in unplanned expenditures”, while the program should have provided “\$70 million in annual savings by centralizing pay operations” [40]. Similarly, bugs in Louvois, the French army payroll system, led to several years of incorrect payments to military households, either causing missed wages or overpayments that had to be reimbursed by individuals years later [42]. Such cases are not isolated incidents, and examples of public legal systems that were either faulty or abandoned during development abound in many countries [47].

One of the core issues lies with the particular structure of computational law, commonly consisting of a base case refined by exceptions spread out throughout law texts. This structure, corresponding to *default logic* [6, 28], is not straightforward to encode using modern programming languages. This leads to discrepancies between code and law, where design choices made by programmers can be hard to understand by lawyers. Conversely, maintaining such implementations when legislative processes modify the law becomes tricky, as programmers cannot necessarily accurately pinpoint required changes in their code.

To address this issue, Merigoux et al. [37] recently introduced Catala, a domain-specific language (DSL) tailored to implement computational laws. At the heart of Catala lies programming constructs called *default terms*, which faithfully implement default logic. Default terms are first-class citizens in the core Catala language. To simplify their use, they are exposed through a custom syntax designed to be understandable by programmers and lawyers alike. This syntax enables Catala programmers to define the base case of a default term and its corresponding exceptions in different parts of the code, thus accurately reflecting the structure of legal texts. By combining this feature with literate programming, Catala enables a programming style where an official legal text is intertwined with its implementation, thus allowing programmers to work in concert with lawyers to ensure a faithful translation into code of the law.

To minimize common programming issues, Catala’s design purposely avoids risky programming languages constructs, e.g., by providing infinite-precision numbers rather than floating-point ones, and by avoiding NULL values. Despite this effort, legal implementations in Catala can nevertheless contain bugs for three reasons. First, by heavily operating on numerical values such as amounts of money, Catala is not immune to standard runtime errors such as divisions by zero, possibly leading to crashes. Second, in spite of its peer-programming

approach where lawyers and programmers work together to implement the law, translations of legal texts into code amount to legal interpretations, which can be erroneous. Last, the law itself might exhibit inconsistencies, for instance, by defining contradicting exceptions under specific circumstances, or by forgetting to consider some situations – in Catala implementations, this would lead to a runtime exception being raised by the program.

To circumvent these issues, public administrations heavily rely on testing; some departments are in charge of manually handcrafting testcases, interpreting the law to compute results, and comparing them to the outputs of legal expert systems to ensure their conformance with legal texts. Unfortunately, the high number of cases, and the regular modifications of laws make this manual process costly and painstaking, and do not guarantee that corner cases are not missed.

To address this problem, this paper therefore advocates applying formal verification techniques to computational law. Doing so raises several challenges. First, formal verification tools must be able to efficiently reason about the core concept of computational law, namely, default logic. Second, to facilitate its adoption, the verification process must not require expert knowledge of formal verification, and thus needs to be as automated as possible. Last but not least, when identifying issues, it is crucial that verification tools provide concrete examples usable by lawyers, so that they can compare them with legal texts and determine whether these correspond to law inconsistencies.

Given these constraints, we focus our attention on the application of concolic unit testing [21, 51], a combination of concrete execution with SMT-based symbolic execution, which we believe to hit a sweet spot when it comes to reasoning about computational law. Computational law, and thus its implementation in Catala, is deterministic, and does not contain loops or recursion, alleviating a well-known challenge for concolic testing [10, 20]. Additionally, its automated nature and the concrete inputs it generates during analysis pave the way for use by lawyers and programmers alike.

Our contributions are the following: we show how to concolically execute computational law programs by providing a formal concolic semantics for default terms (Sec. 2.2). Relying on this formal model, we then implement CUTECat, a concolic execution engine for Catala programs (Sec. 3). CUTECat aims to detect all runtime errors, such as divisions by zero but also law inconsistencies due to missing, or conflicting interpretations. CUTECat includes several optimizations, both to improve its performance and scalability, as well as its usability, aiming to generate human-friendly testcases to facilitate legal interpretation by lawyers (Sec. 4). Finally, we experimentally evaluate CUTECat on a range of Catala programs (Sec. 5). Relying on real-world codebases from French and US laws, we first perform an ablation study to evaluate the impact of our different optimizations. We then conclude by empirically demonstrating that CUTECat can scale to the largest real-world Catala codebase currently available, namely, the implementation of the French housing benefits (19655 lines of Catala, including specification), generating 186390 tests in less than 7 hours of

CPU time. Our development is open-source and publicly available on GitHub; all experimental claims made in this paper are documented in an artifact [23].

2 Encoding Default Logic in Concolic Execution

In this section, we present our encoding of default logic into concolic execution. We start with some background about computational law, and how it relates to default logic.

2.1 Computational Law and Default Logic

Throughout this paper, we will rely on a simplified income tax computation as a running example. This computation determines the amount of taxes a household must pay, depending on their income. Typically, regulations specify a tax rate, that is multiplied by the household income to determine the income tax to pay. For instance, if the tax rate is set to 20%, a household earning \$100,000 would pay \$20,000 in income tax. In practice, tax laws commonly define different tax rates for different income brackets; we omit this in our example for simplicity.

To address several societal issues, this computation can however be modified depending on households. For instance, low-income households (which we define in this example as earning no more than \$10,000) might be taxed to a reduced rate of 10%. Furthermore, large households (which we define in this example as households with three or more children), might benefit from a tax break, for instance, lowering their tax rate to 15%. For now, we do not set any interpretation priority for households having both a low income and a large number of children.

This structure is typical of computational laws: we have one *base case* (the tax rate is 20%), followed by several *conditional exceptions*, capturing specific situations (large, or low-income households). Implementing this structure in traditional programming languages is challenging for two reasons. First, exceptions in legal texts are commonly spread out through several law articles, which does not match the standard monolithic variable or function definitions. Second, legal implementations should closely follow the structure of the law they implement. Implementations correspond to applications of the law; any difference with the original text thus amounts to a legal reinterpretation, which should be performed by lawyers, and not programmers. Heavily transforming the law to match traditional programming idioms is therefore ill-advised. To address this issue, earlier work proposed *default terms* [37], heavily inspired by default logic [6, 28].

Default terms take the form $\langle e_1, \dots, e_n \mid e_{just} :- e_{cons} \rangle$, where $e_1, \dots, e_n$ are default expressions called ‘exceptions’, e_{just} is a boolean expression called ‘default condition’, and e_{cons} is a default expression. Empty values are written as $\emptyset$, and conflict values, when two exceptions occur at the same time, as $\oplus$. Due to typing guarantees, omitted here for brevity, the default condition never reduces to $\emptyset$ or $\oplus$. The grammar and formal semantics of default terms, adapted from [37], are available in Fig. 1 and Fig. 2 respectively. Expressions and values also contain numeric and boolean expressions, whose semantics is standard and

Value	$v ::= \text{true} \mid \text{false} \mid n$	boolean, integer literals
	$\mid \emptyset$	empty value
	$\mid \otimes$	conflict value
Expression e	$e ::= x \mid v$	variable, values
	$\mid e \bowtie e$	integer, boolean binary operators
	$\mid \langle \vec{e} \mid e :- e \rangle$	default term

Fig. 1: Grammar of default terms

DEFAULTEXPR		$\frac{e \xrightarrow{*} v \quad v \neq \otimes}{\langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow \langle v_1, \dots, v_i, v, \dots \mid e_{just} :- e_{cons} \rangle}$	
DEFAULTERROR	$\frac{e \xrightarrow{*} \otimes}{\langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow \otimes}$	DEFAULTTRUENOEXCEPTIONS	$\frac{e_{just} \xrightarrow{*} \text{true}}{\langle \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \longrightarrow e_{cons}}$
DEFAULTFALSENOEXCEPTIONS	$\frac{e_{just} \xrightarrow{*} \text{false}}{\langle \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \longrightarrow \emptyset}$	DEFAULTONEEXCEPTION	$\frac{v \neq \emptyset, \otimes}{\langle \emptyset, \dots, \emptyset, v, \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \longrightarrow v}$
DEFAULTEXCEPTIONSCONFLICT $\frac{v_i \neq \emptyset \quad v_j \neq \emptyset \quad \forall k, v_k \neq \otimes}{\langle v_1, \dots, v_i, \dots, v_j, \dots, v_n \mid e_{just} :- e_{cons} \rangle \longrightarrow \otimes}$			

Fig. 2: Selected reduction rules for default terms

thus omitted from the presentation. According to this semantics, default terms are executed as follows:

- If all exceptions reduce to empty values, that is, if no exception is raised or if there are none, then the default expression reduces either to e_{cons} if condition e_{just} evaluates to true (DEFAULTTRUENOEXCEPTIONS), or to $\emptyset$ otherwise (DEFAULTFALSENOEXCEPTIONS). For instance, the term $\langle \mid \text{false} :- 1 \rangle$ reduces to $\emptyset$.
- If exactly one exception e_i reduces to a non-empty value, then the default expression reduces to its value (DEFAULTONEEXCEPTION). Therefore, the term $\langle \langle \mid \text{true} :- 1 \rangle, \langle \mid \text{false} :- 2 \rangle \mid \text{true} :- 3 \rangle$ reduces to 1.
- If more than one exception expression reduces to a non-empty value, that is if several exceptions are raised at the same time, then the default expression reduces to a conflict error $\otimes$ (DEFAULTEXCEPTIONSCONFLICT)³. Thus, $\langle \langle \mid \text{true} :- 1 \rangle, \langle \mid \text{true} :- 2 \rangle \mid \text{true} :- 3 \rangle$ yields $\otimes$.

Remark 1. To lighten notations, we will omit the list of exceptions from a default term when it is empty. For instance, the term $\langle \mid e_1 :- e_2 \rangle$ will be written as $\langle e_1 :- e_2 \rangle$.

³ Two raised exceptions are considered a conflict even if they lead to the same result.

Encoding the income tax rate we described earlier as a default term is then straightforward. As the base case always applies, the default condition e_{just} is trivial, and the two cases for low-income and large households are encoded as exceptions, leading to the following term:

$$\langle (\text{income} \leq \$10,000 :- 10\%), (\text{nb\_children} \geq 3 :- 15\%) \mid \text{true} :- 20\% \rangle$$

2.2 Concolically Executing Default Terms

We now present how to support default terms during concolic execution. We start with some preliminaries about concolic execution, and the related concept of symbolic execution.

Background: Symbolic and Concolic Execution. Symbolic execution [5, 14, 24, 27] is a program analysis technique that aims to explore all feasible program paths in a program, and that has been successfully applied to a wide range of languages [8, 9, 13, 16, 39, 45, 46]. In symbolic execution, concrete inputs are replaced by symbolic values, and a symbolic interpreter is tasked with executing the program under test. As the program is executed, the interpreter will collect symbolic constraints characterizing a given program execution path through a symbolic *path constraint*. When hitting a conditional branching, the symbolic interpreter will add the condition to the current path constraint, query an SMT solver [2, 4, 15, 18] to determine whether the path constraint is satisfiable, and resume execution of the program. When a constraint is deemed unsatisfiable, the interpreter will backtrack, and attempt to explore another execution path.

To make things more concrete, consider the small program $P \stackrel{\text{def}}{=} \text{if } x > 0 \text{ then return } 0 \text{ else if } y < 10 \text{ then return } 1 \text{ else error}$. We present the different steps of the symbolic execution in Fig. 3, showcasing the generated path constraint tree at different steps. Treating x and y as symbolic variables, a symbolic interpreter would first reach the conditional branching `if  $x > 0$` , and generate two paths to explore, corresponding to the path conditions $x > 0$ and $\neg(x > 0)$ (Fig. 3a). As both path conditions are satisfiable, both execution paths need to be explored. Arbitrarily picking the first path condition, it would then reach the terminal statement `return 0`, terminating the execution of this path (Fig. 3b). Executing the second path, it would then split the path constraint tree again according to the constraint `if  $y < 10$` . As both path conditions $\neg(x > 0), y < 10$ and $\neg(x > 0), \neg(y < 10)$ are satisfiable, the symbolic interpreter would explore both paths, thus detecting that the error statement is reachable (Fig. 3c).

Despite its successes, symbolic execution however faces several limitations. Symbolic execution tools heavily rely on SMT solvers, and therefore struggle when considering symbolic constraints beyond theories well-supported by automated solvers, such as non-linear arithmetic. Additionally, symbolic interpreters typically require access to the analyzed code, and hence do not work well when interacting with an external environment or external library calls [1, 10, 20].

To address these issues, *concolic execution*, a combination of *concrete* and *symbolic* execution was proposed [21, 51]. In concolic execution, the program is

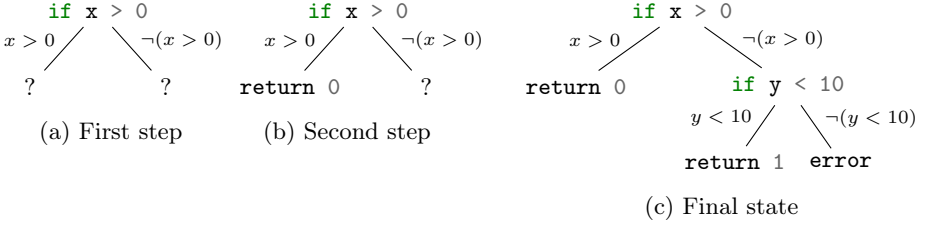


Fig. 3: Path constraint trees at different stages of symbolic execution. Nodes marked as ? are not yet explored

executed with concrete inputs, and instrumented to collect symbolic constraints during execution. When one execution terminates, some of the constraints in the path condition are negated and new inputs are generated through a call to an SMT solver, leading to the exploration of a new program path.

Compared to symbolic execution, concolic execution has several advantages. First, when faced with unsupported logical theories, concolic execution can still make progress by executing the program, instead of getting stuck when encountering complex constraints. Second, by relying on concrete inputs covering a wide range of program paths, concolic execution naturally provides a reproducible testbed that can be used for automated unit testing, even when symbolic execution would be imprecise due to complex program statements or calls to libraries whose code is unavailable [20]. Last, while the generation of new inputs is typically performed by an automated solver to guarantee the exploration of new program paths, it can also be combined with other input generation techniques such as fuzzing [22, 32, 44, 54].

To formally describe concolic execution, we will rely on the following notations for instrumented semantics. When $e \longrightarrow^* e'$ denotes the concrete evaluation from expression e to expression e' , $\chi \vdash e \longrightarrow^* \chi' \vdash e'$ will denote that, under initial path condition χ , e evaluates to e' with new path condition χ' .

Coming back to our toy program P , concolic testing would therefore proceed as follows. Let us assume that we start with inputs $x = 5, y = 5$. Then, we would have $\cdot \vdash P \longrightarrow^* x > 0 \vdash 0$, i.e., the concrete execution of the program returned the value 0, and the path condition $x > 0$ was collected. A concolic engine would then negate part of the path condition, query the SMT solver with the new constraints, i.e., $\neg(x > 0)$, and start another iteration with the inputs generated by the solver. This process would repeat until negating constraints does not lead to novel execution paths and all feasible program paths have been explored, with concrete inputs leading to each of these paths.

Encoding Default Terms. To concolically execute default terms, we now need to provide a symbolic representation to complement the concrete semantics presented in Fig. 2. To do so, we show how to instrument the concrete semantics to collect symbolic constraints characterizing the current execution path, by materializing the control flow structure of the evaluation of a default expression. In order to cover all program paths induced by default terms, this encoding must

therefore consider all combinations of raised exceptions, as well as a branching due to the default condition e_{just} when no exception is triggered.

We formally define our instrumented semantics in Fig. 4. We slightly extend the notations presented in the previous section, and denote as $\chi \vdash e \mid C$ a default expression e with symbolic constraints χ and C . The constraint χ corresponds to the path condition collected up to the evaluation of the current default term. The constraint C , which we will call *local constraint*, corresponds to the constraints collected during the evaluation of the current term. The distinction between χ and C is irrelevant for the rules presented in this section, and $\chi \vdash e \mid C$ can safely be understood as $\chi \wedge C \vdash e$, i.e., the current path condition is the conjunction of χ and C . The importance of this separation will appear in Sec. 4.1, when discussing several optimizations to concolic execution of default terms. When either χ or C are trivial, we will omit them to simplify notations.

To illustrate how our semantics operates, we will rely on the following example, where x and b are respectively integer and boolean variables: $\langle \langle b :- 1 \rangle, \langle x = 0 :- 2 \rangle \mid x > 0 :- 3 \rangle$. Let us assume that we want to concolically execute this term with $x = 3$ and $b = \mathbf{true}$. Initially, both the global constraints χ and local constraints C are trivial. Following rule C-DEFAULTEXPR, we must first reduce the exception $\langle b :- 1 \rangle$ to a value. Since we have a default term (cf. Remark 1) and its default condition b holds, we can apply rule C-DEFAULTTRUENOEXCEPTIONS, obtaining the concolic term $b \vdash 1$. We conclude the application of rule C-DEFAULTEXPR, leading to the concolic term $\langle 1, \langle x = 0 :- 2 \rangle \mid x > 0 :- 3 \rangle \mid b$. We then perform a similar reduction on the second exception by combining rules C-DEFAULTEXPR and C-DEFAULTFALSENOEXCEPTIONS to obtain the concolic term $\langle 1, \emptyset \mid x > 0 :- 3 \rangle \mid b \wedge \neg(x = 0)$. As exactly one exception reduced to a non-empty or error value, the final step in the concolic execution is then to apply rule C-DEFAULTONEEXCEPTION; local constraints are thus appended to the current (empty) path condition. The execution therefore terminates with the value $b \wedge \neg(x = 0) \vdash 1$, accurately capturing that any set of inputs satisfying the path constraint $b \wedge \neg(x = 0)$ will follow the same execution path.

A Complete Concolic Execution. Equipped with our concolic semantics, we can now define a concolic execution operating on default terms. To do so, we follow the standard workflow of concolic execution: starting from an initial, arbitrary set of inputs, we concolically execute the program, and collect the corresponding path constraints. Once the execution terminates, we choose another unexplored path, querying the SMT solver to generate an input satisfying the new path constraint, and therefore leading to a different execution. We repeat this process until all paths have been explored, or an exploration timeout has been reached.

To illustrate how a complete concolic execution operates, we will reuse the default term $\langle \langle b :- 1 \rangle, \langle x = 0 :- 2 \rangle \mid x > 0 :- 3 \rangle$ previously presented. Starting from inputs $x = 3$ and $b = \mathbf{true}$, we previously saw that this term reduced to the value 1, generating the constraints b and $\neg(x = 0)$. By negating the last constraint and querying the solver, we obtain new inputs $x = 0$ and $b = \mathbf{true}$, and perform another concolic iteration. Repeating this process leads to the generation of 5 different testcases fully covering the program, and identifying two

Notations

$\boxed{\chi \vdash e \mid C}$	“Expression e has path condition χ and local constraint C ”
$\boxed{\chi \vdash e}$	“Syntactic sugar for $\chi \vdash e \mid \cdot$ ”
$\boxed{e \mid C}$	“Syntactic sugar for $\cdot \vdash e \mid C$ ”
$\boxed{e}$	“Syntactic sugar for $\cdot \vdash e \mid \cdot$ ”
$\boxed{\chi \vdash e \mid C \longrightarrow \chi' \vdash e' \mid C'}$	“Under path condition χ and local constraint C , e evaluates to e' with new path condition χ' and local constraint C' ”

C-DEFAULTEXPR

$$\frac{e \longrightarrow^* \chi' \vdash v \quad v \neq \otimes}{\chi \vdash \langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle v_1, \dots, v_i, v, \dots \mid e_{just} :- e_{cons} \rangle \mid C \wedge \chi'}$$

C-DEFAULTERROR

$$\frac{e \longrightarrow^* \chi' \vdash \otimes}{\chi \vdash \langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge \chi' \wedge C \vdash \otimes}$$

C-DEFAULTTRUENOEXCEPTIONS

$$\frac{e_{just} \longrightarrow^* \chi' \vdash \text{true}}{\chi \vdash \langle \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \wedge \chi' \wedge e_{just} \vdash e_{cons}}$$

C-DEFAULTFALSENOEXCEPTIONS

$$\frac{e_{just} \longrightarrow^* \chi' \vdash \text{false}}{\chi \vdash \langle \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \wedge \chi' \wedge \neg(e_{just}) \vdash \emptyset}$$

C-DEFAULTONEEXCEPTION

$$\frac{v \neq \emptyset, \otimes}{\chi \vdash \langle \emptyset, \dots, \emptyset, v, \emptyset, \dots, \emptyset \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \vdash v}$$

C-DEFAULTEXCEPTIONSCONFLICT

$$\frac{v_i \neq \emptyset \quad v_j \neq \emptyset \quad \forall k, v_k \neq \otimes}{\chi \vdash \langle v_1, \dots, v_i, \dots, v_j, \dots, v_n \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge C \vdash \otimes}$$

Fig. 4: Concolic semantics for default terms

execution paths leading to $\otimes$ and $\emptyset$ respectively. The summary of the concolic execution is available in Fig. 5.

3 CUTECat: Implementing Concolic Execution

Relying on the default logic and its concolic interpretation presented in the previous section, we now present CUTECat, a concolic execution engine for the Catala programming language.

Catala is a recent domain-specific language tailored to implement computational laws, relying on default logic under the hood [37]. We show an example Catala program in Fig. 6, encompassing the default term for the simplified in-

Step	b	x	Output	Constraints after evaluation	Next constraints to try
# 1	true	3	1	$[(b), \neg(x = 0)]$	$b \wedge x = 0$
# 2	true	0	⊗	$[(b), (x = 0)]$	$\neg b$
# 3	false	3	3	$[\neg(b), \neg(x = 0), (x > 0)]$	$\neg b \wedge \neg(x = 0) \wedge \neg(x > 0)$
# 4	false	-1	∅	$[\neg(b), \neg(x = 0), \neg(x > 0)]$	$\neg b \wedge x = 0$
# 5	false	0	2	$[\neg(b), (x = 0)]$	—

Fig. 5: Concolic testcase generation for $\langle\langle b :- 1 \rangle, \langle x = 0 :- 2 \rangle \mid x > 0 :- 3 \rangle$

```

1  ``catala
2  declaration structure Household:
3    data income content money
4    data nb_children content integer
5
6  declaration scope IncomeTaxComputation:
7    input house content Household
8    internal tax_rate content decimal
9    output income_tax content money
10 ``
11
12 ## Article 1
13 The income tax for an individual is
14 defined as a fixed percentage of the
15 individual's income over a year.
16 ``catala
17 scope IncomeTaxComputation:
18   definition income_tax equals
19     house.income * tax_rate
20 ``
21
22 ## Article 2
23 The fixed percentage mentioned at
24 article 1 is equal to 20%.
25 ``catala
26
27 scope IncomeTaxComputation:
28   definition tax_rate equals 20%
29 ``
30
31 ## Article 3
32 If the individual's income is less
33 than $10,000, the fixed percentage
34 mentioned at article 1 is equal to 10%.
35 ``catala
36 scope IncomeTaxComputation:
37   exception definition tax_rate
38     under condition house.income <= $10,000
39     consequence equals 10%
40 ``
41
42 ## Article 4
43 If the individual is in charge of 3 or
44 more children, then the fixed percentage
45 mentioned at article 1 is equal to 15%.
46 ``catala
47 scope IncomeTaxComputation:
48   exception definition tax_rate
49     under condition house.nb_children >= 3
50     consequence equals 15%
51 ``

```

Fig. 6: Running example: a simplified income tax computation

come tax computation described in Sec. 2.1. One important aspect of Catala is its literate programming capabilities, allowing to reflect the structure of the law in the implementation. As seen in this example, each unit of law is immediately followed by its implementation. The main purpose is to facilitate the maintainability and transparency of the implementation with respect to the law test; at compilation time, Catala will rely on the Markdown code markers ```catala` to extract the implementation. We refer the interested reader to the Catala tutorial [36] for a more detailed description of the merits of literate programming when implementing computational law.

We now provide an overview of the Catala language on our running example. Catala requires strongly typed declarations exposing the interfaces of a program. Here, lines 2-4 define a `Household` record, defined by an `income` and a number of children. Scopes are the basic abstraction unit in Catala; they can be conceived as a loose equivalent to functions in other programming languages. A scope is first statically declared, with a typed declaration of its inputs, outputs and internal variables. In our example, lines 6-9 define the interface of our income tax computation, which computes an income tax for a given household. The main case of the income tax computation is defined lines 17-19 and 26-27: the default

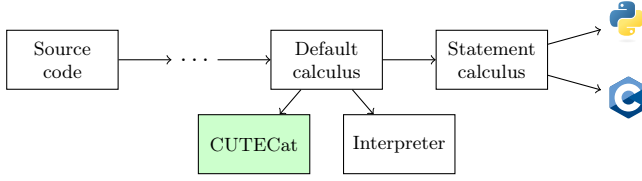


Fig. 7: Passes of the Catala compiler, and CUTECat integration

tax rate is 20% of the household’s income. This rate is amended by two cases, one setting it at 10% for low incomes (lines 35-38), the second setting the rate to 15% for large families (lines 46-49). Note that while these new definitions are following one another in this simplified example, real-world Catala implementations might have amended these definitions in several different files, for instance if the base case is defined in the Tax Code while the large families exception is defined in the Family Code. During compilation, the Catala toolchain will combine the three definitions of `tax_rate` in `IncomeTaxComputation` through the use of a default term, corresponding to the one defined in the previous section: $\langle \langle \text{income} \leq \$10,000 :- 10\% \rangle, \langle \text{nb\_children} \geq 3 :- 15\% \rangle \mid \text{true} :- 20\% \rangle$

Our concolic engine, CUTECat, has been integrated directly into the Catala toolchain, whose relevant parts are shown in Fig. 7. To allow integration of Catala implementations in existing projects, the compiler translates Catala source code into mainstream programming languages like C or Python. To do so, it relies on a series of intermediate representations (IRs), elided here, until reaching the default calculus. The default calculus is one of the last IRs in the compilation pipeline, and is very close to the statement calculus which transforms expressions into statements before emitting C or Python code.

Default terms are explicitly materialized in the default calculus; it is where the reference Catala interpreter operates, following the semantics for default terms described in Fig. 2. This is therefore a natural choice for the implementation of our concolic engine; additionally, being located at the same compilation step of the interpreter nullifies any discrepancies of potential bugs introduced earlier in the compilation pipeline. We implemented our concolic interpreter as a fork of the standard interpreter, instrumenting it to collect symbolic constraints. This design allows us to closely follow the reference semantics of Catala, thus minimizing implementation mistakes, while also simplifying the maintenance of CUTECat when the Catala language evolves.

Similarly to other works on concolic execution [32, 51], CUTECat relies on a depth-first search (DFS) strategy to explore new execution paths. In a DFS exploration, a concolic engine generates new inputs by negating the last constraint added to the path condition. Concretely, during a concolic iteration, CUTECat keeps the constraints C used to generate the current input. Each constraint is annotated to indicate whether it has already been negated. The execution generates a path condition C' , which is compared to C ; all constraints in $C' \setminus C$ are marked as new, and the last new constraint in C' is negated to generate a new input. This workflow corresponds to a DFS exploration of the path constraint tree;

```

1  scope IncomeTaxComputation:
2    exception definition income_tax
3    under condition house.income <= $10,000
4    consequence equals house.income * 10%

```

Fig. 8: Removing conflicts in income tax computation

however, it only requires keeping the last explored branch in memory instead of the entire tree, therefore improving the performance of concolic execution.

To illustrate how CUTEcAt operates, we now describe the concolic execution of our running example. CUTEcAt generates here four different cases, in an order determined by the values used in the first iteration; in our case, they are set to `income = $0` and `nb_children = 0`.

1. `income` is \$0 and there are no children. In that case, only the first exception (lines 35-38 of Fig. 6) can be applied, and the tax is \$0.
2. `income` is \$0 and there are three children. In that case, both exceptions (lines 35-38 and 46-49) can be applied. A conflict is raised to the user, showing that the law has been unfaithfully translated, or that it is ambiguous.
3. `income` is \$10,000.01 and there are two children. In that case, the general case (lines 26-27) applies, and the tax is \$2,000 (monetary amounts are rounded to the cent).
4. `income` is \$10,000.01 and there are three children. In that case, the exceptional case at lines 46-49 applies and the tax is \$1,500.

By default, CUTEcAt outputs these different cases directly to the user, however, we also provide facilities to directly generate Catala *test scopes*. These test scopes can be saved, and easily replayed using the Catala toolchain, either by using the reference interpreter or by compiling it to one of the Catala backends.

As identified by our concolic execution, our running example can raise conflicts, i.e., when considering large, low-income households. This conflict is not due to an implementation error, but rather to an ambiguity in our (simplified) law. Resolving this ambiguity requires legal interpretation, which must be performed by lawyers, and not programmers. In this case, legal precedents might suggest that ambiguities shall be resolved in the most favorable way to citizens, therefore defining a 10% tax rate. To implement this decision, programmers could thus modify the Catala implementation of Article 3 as shown in Fig. 8, by materializing a priority order on the evaluation of both exceptions: as the exception is now defined on `income_tax`, this exception will trigger first, before reaching the base computation depending on `tax_rate`⁴.

CUTEcAt is implemented in 3,400 lines of OCaml code, and relies on the Z3 SMT solver [18], called through its OCaml bindings. It is an integral part of the Catala toolchain, and therefore leverages its build system, simplifying the use of CUTEcAt on Catala projects. To provide users with more control on the concolic execution, CUTEcAt allows the use of assertions in the source code to restrict

⁴ Note that a more idiomatic fix would make use of Catala "labels", that enable programmers to order exceptions explicitly. We omit this solution to keep our presentation of Catala small. The Catala tutorial [36] has more information on labels.

the input space, for instance, to specify that only households in a given country or state must be considered. Users can provide hints about initial inputs for the concolic execution, in order to quickly explore variants of a given situation.

Divisions and Rounding. In addition to our handling of default terms described in Sec. 2, CUTECat supports the concolic execution of most Catala constructs, including arbitrary-precision integers, booleans, money and decimal expressions; structures and field operations; algebraic data types and pattern-matching; functions and function calls; and conditionals. The concolic execution of most of these operations is standard, and we therefore omit their presentation. Two points are however of particular interest, namely, arithmetic divisions and rounding.

Divisions by zero are a well-known source of runtime errors. To reason about them, concolic tools therefore consider divisions as implicitly branching, and collect whether the denominator is equal to zero as a symbolic constraint.⁵

By providing different representations for numerical values, i.e., decimal (represented as rationals) and integers values, Catala also requires conversions between these types; when casting a decimal to an integer, this is implemented as a rounding operator. Catala’s rounding convention is to round to the nearest integer; when two are equidistant, it returns to the one furthest away from 0. To model this semantics, we define a custom *round* Z3 function, defined as `round(q) = if q>=0 then floor(q + 1/2) else -floor(-q + 1/2)`; the `floor` operation is implemented using Z3’s native casting from rationals to integers.

As part of our experimental evaluation of CUTECat (Sec. 5), our precise modeling of rounding led us to find inputs where the different Catala backends were inconsistent: rounding operations in Catala’s Python backend exhibited minor differences with the reference interpreter. We reported this issue to the Catala developers, and upstreamed a fix to the compiler.

Limitations. CUTECat currently supports a large subset of Catala which is sufficient to analyze real-world programs (Sec. 5) that mainly rely on integer-rational reasoning as well as algebraic datatypes. However, operations on lists and dates are only partially supported. List operations in Catala include filtering depending on a condition, aggregating over list contents to compute all sorts of values, and applying a function to all elements in the list; providing symbolic encoding of such operations would require higher-order symbolic reasoning [43, 55]. On the other hand, dates can easily be represented in an SMT solver as the number of days since a specific point in time, e.g., the Unix epoch. This representation allows to model several operations, such as the addition of a (possibly symbolic) number of days, or the duration corresponding to the difference between two

⁵ Note that our semantics conflates runtime errors with conflicts, and returns $\otimes$ in both cases. Indeed, the conflict value acts as a *de facto* uncatchable exception: when it appears, it stops the evaluation of the program, which immediately returns. From the user’s perspective, the main concern is whether the program can lead to an error, independently of which one. As we generate concrete inputs, it is always possible to run the reference Catala interpreter to provide precise error reporting.

dates. Unfortunately, other operations, including month and year addition or returning the first day of a week or a month, have a more complex semantics [41] which does not naturally fit into this SMT encoding.

To handle these operations, we leverage the strengths of concolic execution over symbolic execution, and do not generate any symbolic encoding, relying instead entirely on concrete evaluation at the cost of completeness. We leave the exploration of suitable symbolic models for these operations to future work.

Finally, our implementation currently only supports the Z3 SMT solver, as it directly provides ready-to-use OCaml bindings. This limitation could be easily lifted, as we can already generate SMT-LIB files [3] for each SMT query, which can be consumed by most other SMT solvers [2, 4, 15]. Additionally, our prototype is single-threaded; the recent introduction of Multicore OCaml [52] nevertheless paves the way for parallelizing concolic execution [7, 53].

4 Improving the Scalability and Usability of CUTECat

To improve the scalability of CUTECat, we now discuss various optimizations and heuristics to enhance the performance of concolic execution, as well as the usability of our toolchain by non-expert users. The concrete impact of these optimizations will be evaluated experimentally in Sec. 5.

4.1 Optimizing Concolic Execution of Default Terms

The approach presented in Sec. 2.2 allows us to perform concolic execution on default terms. However, to improve interactivity with developers, our goal is not only to exhaustively analyze all program execution paths, but also to find possible errors as early as possible during the analysis. In this section, we therefore propose several optimizations aiming to prune constraints leading to redundant cases, and to prioritize executions that might lead to conflict errors.

Lazily Evaluating Exceptions. When evaluating a default term, the semantics presented in Fig. 2 requires first evaluating all exceptions, except if one of them raises a conflict error (`DEFAULTEXPR`). If two exceptions evaluate to a non-empty value, the ensuing conflict is therefore only detected when applying rule `DEFAULTEXCEPTIONSCONFLICT`. As concrete executions are in practice very fast, this has little performance impact when concretely executing a default term; furthermore, from a usability perspective, it is helpful to report all conflicting cases to users. However, the overhead becomes larger during concolic execution, where we must explore all possible execution paths when evaluating remaining exceptions, including many calls to an SMT solver.

To circumvent this issue, we propose in Fig. 9 alternative semantic rules to replace `DEFAULTEXPR` and `DEFAULTEXCEPTIONSCONFLICT`, as well as their concolic counterparts. These new rules stop the evaluation as soon as two exceptions return a non-empty value. This prevents exploring the remaining exceptions; the multiple execution paths would all lead to a conflict in the current default term.

$$\begin{array}{c}
\text{DEFAULTEXPR-LAZY} \\
\frac{e \xrightarrow{*} v \quad v \neq \otimes}{\langle \emptyset, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow \langle \emptyset, \dots, \emptyset, v, \dots \mid e_{just} :- e_{cons} \rangle} \\
\\
\text{DEFAULTEXPRONE-LAZY} \\
\frac{e \xrightarrow{*} \emptyset \quad v_i \neq \otimes, \emptyset}{\langle \emptyset, \dots, v_i, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow \langle \emptyset, \dots, v_i, \dots, \emptyset, \emptyset, \dots \mid e_{just} :- e_{cons} \rangle} \\
\\
\text{DEFAULTEXCEPTIONSCONFLICT-LAZY} \\
\frac{e \xrightarrow{*} v \quad v \neq \otimes, \emptyset \quad v_i \neq \otimes, \emptyset}{\langle \emptyset, \dots, v_i, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow \otimes} \\
\\
\text{C-DEFAULTEXPR-LAZY} \\
\frac{e \xrightarrow{*} \chi' \vdash v \quad v \neq \otimes}{\chi \vdash \langle \emptyset, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle \emptyset, \dots, \emptyset, v, \dots \mid e_{just} :- e_{cons} \rangle \mid C \wedge \chi'} \\
\\
\text{C-DEFAULTEXPRONE-LAZY} \\
\frac{e \xrightarrow{*} \chi' \vdash \emptyset \quad v_i \neq \otimes, \emptyset}{\chi \vdash \langle \emptyset, \dots, v_i, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \vdash \langle \emptyset, \dots, v_i, \dots, \emptyset, \emptyset, \dots \mid e_{just} :- e_{cons} \rangle \mid C \wedge \chi'} \\
\\
\text{C-DEFAULTEXCEPTIONSCONFLICT-LAZY} \\
\frac{e \xrightarrow{*} \chi' \vdash v \quad v \neq \otimes, \emptyset \quad v_i \neq \otimes, \emptyset}{\chi \vdash \langle \emptyset, \dots, v_i, \dots, \emptyset, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge \chi' \wedge C \vdash \otimes}
\end{array}$$

Fig. 9: Alternative concrete and concolic lazy semantics for default terms

Consider for instance a default term $e := \langle e_1, e_2, e_3 \mid e_{just} :- e_{cons} \rangle$, where e_1 and e_2 both evaluate to values that are neither empty nor a conflict, respectively generating constraints C_1 and C_2 . By following rule C-DEFAULTEXPR, concolically executing e would require concolically executing e_3 and generating its associated constraint C_3 , which would then be added to the path condition when applying rule C-DEFAULTEXCEPTIONSCONFLICT. Further iterations of concolic execution would therefore negate the last constraints in the path condition, i.e., those in C_3 , while preserving C_1 and C_2 . As any input satisfying $C_1 \wedge C_2$ leads to e_1 and e_2 evaluating to non-empty nor conflict values, thus raising a conflict, this approach would induce a number of redundant iterations that is exponential in the number of constraints in C_3 . By instead adopting rule C-DEFAULTEXCEPTIONSCONFLICT-LAZY and stopping the execution after the evaluation of e_2 , we therefore prune C_3 from the path condition, thus reducing the number of iterations needed.

Reorganizing Exceptions. By relying on the semantics presented in Fig. 9, concolic executions of default terms can therefore be greatly shortened if exceptions evaluating to non-empty values are at the front of the exception list. Beyond not evaluating additional exceptions, their corresponding symbolic constraints will not be added to the path condition, thus reducing the size of the constraint tree and the number of iterations needed for concolic execution to terminate.

To leverage this fact, one key observation is that the result of the evaluation of a default term is independent of the evaluation order of the exception list. We formalize this property in Th. 1, which states that swapping any two exceptions

$$\frac{\text{C-DEFAULTERROR-EARLY} \quad e \longrightarrow^* \chi' \vdash \circledast}{\chi \vdash \langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C \longrightarrow \chi \wedge \chi' \vdash \circledast}$$

Fig. 10: Alternative concolic semantics for conflict-inducing evaluations

results in evaluating the default term to the same value. This therefore allows to use different evaluation orders depending on the program and current symbolic state. In our implementation, which we describe in more detail in the next sections, we permute exceptions to group them depending on their free variables, a technique reminiscent of constraint reordering introduced by Cadar et al. [8].

Theorem 1 (Independence of the exception evaluation order). *If there exists a default value v such that $\langle \dots, e_i, \dots, e_j, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow^* v$, then $\langle \dots, e_j, \dots, e_i, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow^* v$*

Pruning Non-Conflict Path Conditions. As a last optimization, we present in Fig. 10 an alternate formulation of rule C-DEFAULTERROR, which applies when evaluating an exception yields a conflict value. If $\chi \vdash \langle e_1, \dots, e_i, e, \dots \mid e_{just} :- e_{cons} \rangle \longrightarrow^* \chi \vdash \langle v_1, \dots, v_i, e, \dots \mid e_{just} :- e_{cons} \rangle \mid C$, and $e \longrightarrow^* \chi' \vdash \circledast$, then the application of rule C-DEFAULTERROR will add both C and χ' to the path condition. We observe however that this will lead to redundant executions, as any input satisfying χ' will lead to e evaluating to $\circledast$. Pruning C from the path condition thus avoids unneeded concolic exploration, which we formally capture through rule C-DEFAULTERROR-EARLY.

This rule directly derives from Th. 1; indeed, the rule C-DEFAULTERROR-EARLY is conceptually equivalent to reorganizing the exception list to place e at the head of the list, and then applying C-DEFAULTERROR. This is where the need for differentiating between the path condition χ and local constraint C arises: while constraints corresponding to the evaluation of earlier exceptions can be safely dropped, the path condition up to this default term must be preserved.

Implementation. CUTEcat implements the lazy evaluation of exceptions, as well as exception reorganization to order exceptions according to their sets of free variables. Pruning non-conflict path conditions is trickier to implement while following a DFS exploration strategy. A key invariant of DFS exploration is that, during a concolic iteration, the previous path condition (used to generate the current input) is a prefix of the current path condition up to the negated constraint. This does not hold with rule C-DEFAULTERROR-EARLY, as the local constraint C is dropped from the path condition. We leave the study of alternative exploration strategies and their impact as future work.

4.2 Generating Human-compatible Testcases

While many testcases might be equivalent from a semantic perspective (i.e., following the same program path), some might be easier to review by lawyers

and humans in general. Consider for instance the constraint $\neg(x \leq \$10,000)$, used in our running example to determine whether a household is in the low-income category. As we have seen in Sec. 3, Z3 might generate $x = \$10,000.01$ as a testcase. When crafting testcases, lawyers would instead lean towards round numbers, e.g., $x = \$11,000$, which would be easier to use when redoing manual computations of legal statutes to compare them to the implementation’s output. To mimick this behavior, CUTECat therefore provides heuristics attempting to generate semantically equivalent, “human-friendlier” testcases.

To do so, we rely on the encoding of optional constraints in the solver, dubbed “soft constraints” [59]. Z3 natively allows to combine SMT reasoning with solving optimization objectives, including the specification of soft constraints; however, we encountered significant slowdowns when attempting to use this feature, raising scalability issues, that have been reported to Z3 developers. We posit that this is due to non-linearity in our soft-constraints – e.g., when specifying $x\%100 = 0$ – and in our rounding function defined in Sec. 3. We instead adopt an alternative approach, by implementing custom soft constraints through multiple queries to the solver. Concretely, when an iteration of concolic execution terminates and a new path condition C is generated, we first query Z3 to check whether C is satisfiable. If so, we further query Z3 by iteratively refining C with additional constraints forcing human-friendlier input generation.

Our current soft constraints particularly target monetary inputs. We first attempt to force such inputs to be multiples of 100. If impossible, we then try with multiples of 10, and finally with integers, aiming to prevent the use of cents. Empirically, these constraints have a significant impact: as monetary amounts are encoded at the cent level, Z3 frequently returns instances with cents, which are less readable and less pleasant to use when computing by hand, especially when percentages and rounding are involved. This paves the way for further improvements to CUTECat’s usability; beyond monetary inputs, we envision the development of a lawyer-friendly custom configuration language, enabling the specification of preferred soft constraints depending on needs and usecases.

4.3 Pattern-Matching Case Folding

The Catala language supports simple pattern matching on the constructors of algebraic datatypes. By default, CUTECat’s interpretation of pattern matching considers each case of the pattern matching as a different branch. That way, the logic of the pattern matching is materialized in the constraint tree itself, and the engine is tasked with successively trying each branch. However, we have noticed that this approach can be highly inefficient in existing Catala codebases.

Consider for example the snippet in Fig. 11, extracted from the Catala implementation of the French housing benefits. At lines 5-13, this program performs a match on the 9 variants of a sum type representing geographic areas of France (mainland, or overseas territories). To analyze this program, a naive implementation of our concolic interpreter would therefore consider 9 different, disjoint branches, leading to 9 distinct iterations.

```

1  scope HousingBenefitsForRentals:           8      -- Mayotte: true
2  exception definition family_rate           9      -- SaintBarthélemy: true
3  under condition                           10     -- SaintMartin: true
4  (match area with                          11     -- Guyane: false
5  -- Guadeloupe: true                       12     -- Métropole: false
6  -- Martinique: true                      13     -- SaintPierreEtMiquelon: false)
7  -- LaRéunion: true                       14     consequence equals ...

```

Fig. 11: French housing benefits case where pattern case folding is beneficial

Here however, the arms of several cases are identical; patterns can be conjoined to only create 2 branches, one corresponding to cases returning `true`, the other to cases returning `false`. Doing so heavily reduces the size of the path constraint tree; in more realistic examples, performance gains can be significant when such patterns appear deep inside expressions, requiring long concolic executions. We implemented a case folding optimization for pattern matching inside CUTEcAt, folding similar cases by considering the disjunction of their patterns.

4.4 SMT-Solving Optimizations

Trivial constraint simplification. The path constraints taken into account during the concolic execution can sometimes be trivial. Indeed, some conditions do not depend on input variables, or are compilation artifacts from the translation of the source code into the default calculus, where tautological conditions are sometimes added in default expressions. Negating these trivially true constraints in the hope of exploring other execution paths is pointless, as the new constraint given to the solver will be trivially false, and therefore unsatisfiable.

To avoid this, we query the SMT solver to simplify constraints before they are added in the path, and then remove trivial constraints statically. This avoids spurious calls to the SMT solver. Additionally, we perform a linear check for trivial unsatisfiability, by checking whether the newest constraint is a negation of a previous one. This last technique is well known by the concolic solver community, and already mentioned in the seminal work of Sen et al. [51].

Incremental SMT solving. Z3 provides incremental solving capabilities, where sets of constraints can be incrementally added or removed through stack-based `push` and `pop` operators. When popping the stack, the solver backtracks to an earlier proof state, only removing lemmas learned between `push` and `pop` and avoiding reproving many facts. We implemented support for incremental solving, pushing a new stack for each element in the path condition. As CUTEcAt’s exploration strategy is currently fixed to depth-first search (DFS), incremental solving is particularly appealing: constraints are naturally changed in a last-in-first-out fashion, corresponding to a stack-based data structure.

5 Evaluating CUTEcAt

We now turn to our experimental evaluation of CUTEcAt. We first quantify the impact of the optimizations described in Sec. 4, relying on four Catala projects

predating this work (Sec. 5.1). We then discuss CUTECat’s ability to identify conflicts (Sec. 5.2), before evaluating CUTECat on the largest Catala program up to this day, namely, an implementation of the French Housing Benefits (Sec. 5.3). All experiments have been performed on a desktop machine featuring an Intel Core i7-12700 and 128GB of DDR5 RAM (although we have never seen CUTECat use more than 1GB), running Ubuntu 24.04.1, OCaml 4.14.1 and Z3 4.12.5.

5.1 Evaluating CUTECat’s Optimizations

Benchmarks Overview. To evaluate the performance of CUTECat and its optimizations, we have selected four different, real-world Catala programs as benchmarks for our evaluation. These programs predate our implementation of CUTECat, and have been written by third-party, experienced Catala developers. One benchmark corresponds to the implementation of a fragment of Section 132 of the US Tax Code, defining qualified employee discounts. Other benchmarks implement diverse bodies of law in France: one codifies minimum salary (SMIC), another the computation of a family quotient used in the income tax computation (Family Quotient). The last benchmark computes the housing benefits in the rental case, where additional restrictions have been made: households are located in mainland France and contain less than 10 children. These benchmarks reflect multiple facets of the law, and therefore exhibit diverse encodings in Catala; we provide quantitative details about them in Table 1, namely, the lines of Catala code (including legal specification), the number of branching constructs they contain (default terms, if-then-else, pattern matches), and the size of the compiled Python code. We particularly observe that the default terms outnumber the other branching terms, arguing in favor of their importance.

Evaluation Metrics. To evaluate the completeness of concolic tools, a standard approach is to measure the code coverage achieved. In our context, this is however hard to perform: the Catala interpreter does not provide any coverage measurements, while results on compiled Catala code, e.g., in Python, are highly unreliable. Indeed, on a small program for which we can manually determine all possible execution paths, a single Python execution reaches a line coverage of 65%, while executing all program paths barely reaches 80%. We suspect that this is due to a bloated translation of default logic into mainstream languages. Our evaluation therefore focuses on analysis times, and on the number of testcases generated. We defer a discussion of the completeness of CUTECat to Sec. 5.2.

Solver Calls and Generated Tests. We compare in Table 2 the number of solver calls performed and the number of tests generated by CUTECat. These numbers are provided both for naive executions without any optimization enabled and when all optimizations are enabled. The difference in generated tests evaluates the impact of pruning redundant exploration paths through pattern matching case folding; the difference in solver calls represents the additional impact of our optimizations targeting the SMT encoding, i.e., our handling of trivial constraints. While these optimizations lead to improvements on almost all

Benchmark	Section 132	SMIC	Family Quotient	Housing Benefits
Catala LOC	133	368	776	19655
nb(default terms)	15	17	46	251
nb(if-then-else)	3	0	24	171
nb(match)	2	0	14	139
Python LOC	343	586	1413	31826

Table 1: Statistics about the benchmarks used in the ablation study

Category	Section 132		SMIC		Family Quotient		Housing Benefits	
Optimizations	None	All	None	All	None	All	None	All
Solver Calls	41	24	138	138	13224	4142	355848	126147
Generated Tests	10	10	17	17	381	381	24995	17435

Table 2: Impact of optimizations on solver calls and generated tests

Options	Section 132		SMIC		Family Quotient		Housing Benefits	
	Total	Solve	Total	Solve	Total	Solve	Total	Solve
Standard	0.27s	0.23s	1.01s	0.85s	82.61s	69.46s	6002.03s	2262.38s
Incremental	0.02s	0.01s	0.08s	0.01s	5.21s	0.23s	4306.72s	69.88s

Table 3: Solver and total execution times, with and without incremental solving.

benchmarks (SMIC has no matches or trivial constraints, so it is not affected), their precise impact heavily differs. Avoiding calls to solver when considering trivial constraints is almost always beneficial; it reduces the number of calls up to 3.2x for the family quotient benchmark. Conversely, pattern matching folding only helps the housing benefits; other, smaller examples do not rely as much on large pattern matching. The tables below provide a more detailed analysis of the impact of individual optimizations.

Incremental Solving. We evaluate in Table 3 the gains provided by Z3’s incremental mode (Sec. 4.4). To do so, we compare both the total execution times and the solver execution times when running CUTECat. Similarly to other works [51], our empirical results suggest that incremental solving is a cornerstone of efficient concolic execution; it provides more than an order of magnitude improvement to the solving times. On the smaller examples, these order-of-magnitude improvements are also observed on the total running time of the concolic engine, as the SMT solving dominates the running times. On larger programs, e.g., housing benefits, CUTECat’s execution time is however dominated by running the concolic interpreter; solving constraints only represents 35% of the total time. On this example, incremental solving is 32x faster than Z3’s default mode, leading to a total gain of more than 35 minutes.

Ablation Study. To evaluate the impact of each evaluation, we perform an ablation study and compare total running times for each optimization individually

Optimizations	Section 132	SMIC	Family Quotient	Housing Benefits
None	0.27s $\pm$ 0.00	1.01s $\pm$ 0.01	82.61s $\pm$ 0.55	6002.03s $\pm$ 19.46
Lazy-default	0.27s $\pm$ 0.00	1.02s $\pm$ 0.01	84.31s $\pm$ 1.15	5995.01s $\pm$ 15.21
Exception packing	0.27s $\pm$ 0.00	0.98s $\pm$ 0.01	83.41s $\pm$ 1.58	6013.71s $\pm$ 22.12
Incremental	0.02s $\pm$ 0.00	0.08s $\pm$ 0.00	5.21s $\pm$ 0.04	4306.72s $\pm$ 77.78
Trivial	0.09s $\pm$ 0.00	0.99s $\pm$ 0.01	12.62s $\pm$ 0.11	4258.93s $\pm$ 22.88
Match folding	0.21s $\pm$ 0.00	0.99s $\pm$ 0.01	46.89s $\pm$ 0.54	4045.87s $\pm$ 21.30
Frontend opts.	0.17s $\pm$ 0.00	0.99s $\pm$ 0.00	38.20s $\pm$ 0.42	3538.32s $\pm$ 22.37
All	0.02s $\pm$ 0.00	0.08s $\pm$ 0.00	4.34s $\pm$ 0.07	2843.01s $\pm$ 15.02
Speedup	1250.00%	1162.50%	1803.46%	111.12%

Table 4: Ablation study for CUTECat optimizations. Average on eight runs, with standard deviation displayed after $\pm$.

against the bare implementation of CUTECat (without any optimization), and all optimizations enabled at once. Results are presented in Table 4. We consider the following optimizations: lazy-default and exception packing (Sec. 4.1), incremental solving (Sec. 4.4), constraint simplifications (Sec. 4.4), pattern matching case folding (Sec. 4.3), and frontend optimizations already offered by the Catala compiler. They currently consist in partial evaluation of booleans, and simplification of trivial defaults, i.e., when there are no exceptions and the default condition is a boolean value.

We observe that almost all optimizations provide noticeable performance improvements, but with variability on benchmarks likely tied to variance in their structure. For smaller programs where solving time dominates, the largest improvements are due to incremental solving.

For housing benefits however, other optimizations reduce the total time by an additional 25%. To better describe this example, we measure in Fig. 12 the evolution of the number of tests generated through time. Findings include several results previously observed: the optimizations prune the exploration tree by removing redundant cases, and incremental solving significantly improves total execution time. Additionally, we observe a linear generation rate, suggesting that concolic execution does not struggle to reach certain paths, which could have been a shortcoming of less systematic approaches, e.g., black-box fuzzing.

Notably, optimizations related to default terms, i.e., lazy default and exception packing do not impact the running times of CUTECat. This is however expected: these optimizations are only of interest when terms can return conflict values. Our benchmarks only contain mature Catala programs, corresponding to implementations of enacted laws. To the best of our knowledge, no corpus of conflict-inducing Catala examples currently exists, making a precise evaluation of these optimizations difficult. We however expect these optimizations to shine when applied during development processes, as well as to identify possible inconsistencies during the preparation of new legislation.

Soft Constraints. We evaluate the impact of using soft constraints in Table 5; the evaluation is performed with all optimizations enabled. We only consider

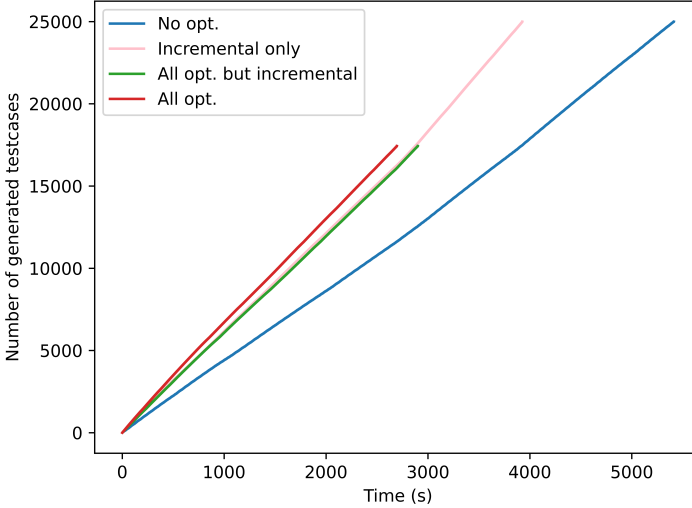


Fig. 12: Comparison of test generation rates on the housing benefits benchmark

Benchmark	Section 132	Housing Benefits
All	0.02s $\pm$ 0.00	2843.01s $\pm$ 15.02
Soft. cons.	0.03s $\pm$ 0.00	2968.80s $\pm$ 16.90
Slowdown	50.00%	4.42%
Number of tests	10	17435
% sat soft cons. (multiples of 100)	100%	87.04%
% sat soft cons. (multiples of 10)	N.A.	9.98%
% sat soft cons. (multiples of 1)	N.A.	2.80%

Table 5: Impact of soft constraints

the Section 132 and housing benefits benchmarks, as our soft constraints target monetary amounts given as inputs, which other examples do not have. On large, representative Catala examples, we observe that the computational overhead of using soft constraints is small. Additionally, the soft constraints are satisfied for almost all cases, significantly improving the usability of CUTEcAt. Indeed, 87.04% of the 17435 generated testcases can be solved with a soft constraint ensuring monetary amounts are multiples of 100, and additionally 9.98% and 2.80% for the multiples of 10 and units, respectively. In total, only 30/17435 generated testcases (0.17%) do not satisfy any soft constraint. Our encoding of soft constraints particularly benefits from incremental solving: they are only added to a path condition proved satisfiable by the solver – which corresponds to refining an existing solution. Furthermore, incremental solving significantly reduces solving time, minimizing the impact on CUTEcAt’s running time of calling the solver several times on a given path. As such, we expect further extensions to soft constraints to have a minor impact on CUTEcAt’s performance.

5.2 Detecting Errors with CUTECat

We now aim to determine whether CUTECat is effective in detecting issues in computational law implementations. Unfortunately, existing Catala codebases are mature and implementing real-world, enacted laws; they are therefore highly unlikely to exhibit such issues. To evaluate CUTECat’s completeness, we therefore turn to mutation testing [26] to automatically inject errors in Catala programs, thereby simulating coding mistakes or legislative imprecisions.

We apply mutation testing to our most complex example: housing benefits. To do so, we first randomly select a default term in the program; we then either remove a chosen number of exceptions (possibly leading to unhandled cases), duplicate an exception (thus creating a conflict), or negate the default condition of a default term (possibly leading to unhandled cases). We follow this process to generate 20 new programs, which we manually inspect to confirm that mutations introduced issues. In particular, we must ensure that the default term considered is reachable, and that assumptions and conditions in other parts of the code do not preclude the unhandled cases, or the execution of the conflicting exceptions.

CUTECat successfully identifies all 20 issues, providing concrete inputs to identify problematic cases. On average, CUTECat requires 0.55 seconds to reach the injected bug; this execution time leads us to believe that our tool would be particularly suitable as part of continuous integration workflows, but also to evaluate intended amendments to laws.

5.3 Case Study: Rental Housing Benefits

Our evaluation so far focused on a constrained case of the rental housing benefits, considering only households located in mainland France and with less than 10 children. We chose to enforce these restrictions due to the multiple executions required to evaluate different CUTECat settings, which already required more than one hour each. In this section, we now explore the general, unconstrained case of the rental housing benefits. With all optimizations enabled, CUTECat is able to analyze this program in 6h37m, generating 186390 testcases. This process generates 1338575 solver calls; the total solver time is 366s.

Interestingly, CUTECat’s analysis led to the discovery of a conflict error in the existing implementation. The housing benefits are defined for various cases, including when the flat is shared with other roommates, or when a single bedroom is rented. However, an interpretation conflict can happen for people claiming rental housing benefits if they are sharing a bedroom with roommates. This conflict was previously manually discovered by Merigoux [35] during their implementation of the housing benefits. Despite querying relevant public administrations, this case was deemed unlikely to happen, and no legal interpretation was provided to resolve this ambiguity. No other errors have been found by CUTECat on the rental housing benefits.

A manual inspection of the generated testcases revealed that the current input space for the housing benefits is under-constrained. For example, some testcases correspond to households located overseas and in housing zone 3 – but

to the best of our knowledge, all overseas territories are defined to be in housing zone 2. Such cases exemplify where expert knowledge from lawyers would be beneficial: provided with indications about which inputs are well-formed, programmers can then restrict CUTECat’s search space through program assertions, therefore speeding up the analysis.

Overhead of Concolic Execution. We now compare the overhead of the concolic execution of CUTECat with the interpretation times of all the generated testcases. The reference Catala interpreter evaluates the 186390 generated tests in 1h29, meaning CUTECat has a 4.5x overhead. While large, this overhead is not unexpected: the concolic interpreter instruments each expression evaluation to collect symbolic constraints, and performs repeated calls to the SMT solver. Prior work on concolic execution observed similar trends: for example, Yun et al. [58] measure that KLEE [8] has up to a three order-of-magnitude overhead, while our experimental results are in line with approaches focusing on reducing this overhead [12, 46]. This leads us to believe that CUTECat’s scalability is comparable to concolic tools in other languages; we nevertheless intend to perform a deeper profiling of our interpreter to identify potential bottlenecks. This comparison also allows us to sanity-check our implementation with respect to Catala’s semantics: we confirmed that CUTECat and the Catala reference interpreter agreed on all generated testcases. Furthermore, we can use those same testcases to check whether all Catala backends have the same behavior on the case study.

6 Related Work

Concolic Execution. Concolic execution was introduced by the seminal works of Godefroid et al. [21] and Sen et al. [51] almost two decades ago. Since then, it has been applied to various programming languages [8, 9, 31, 34, 50]. We refer the reader to the survey of Baldoni et al. [1] for an extensive coverage of concolic and symbolic execution. As Catala has different features compared to traditional, imperative programming languages, most of the advanced techniques mentioned in the survey (memory encoding, loop summarization) are not relevant for our work. Similarly, human-readability of generated tests is usually not an objective of traditional concolic execution engines. However, Baldoni et al. [1] mention that “the way high-level switch statements are compiled can significantly affect the performance of path exploration”, echoing in an imperative setting the need for an encoding similar to the pattern-matching case folding presented in Sec. 4.3. Giantsios et al. [19] support pattern matching through a compilation to a decision tree, which is the approach used to generate machine code [33]. In Sec. 4.3, we chose to implement a simpler approach as Catala currently supports matching against a single sum type. As we have mentioned in Sec. 5.2, our mutation approach is lacking information to be reliable in user-constrained cases, which required us to manually inspect mutated programs. To create an analysis-oriented benchmarking suite for Catala programs, an interesting avenue would be to rely on the program generation techniques of Vikram et al. [56].

Formal Methods for the Law. Several approaches were previously proposed to reason about default logic. Risch [48] and Cassano et al. [11] both define tableaux-based approaches [25] to reason about propositional formulas expressed in default logic, while Schaub [49] and Linke and Schaub [30] propose query answering methodologies for default logic propositions. By focusing on the underlying logic, these works differ from our approach tailored to program verification, which emphasizes testcase generation and requires reasoning about numerical expressions and program constructs such as structures or enumerations. In their compiler for the French tax code, Merigoux et al. [38] have relied on coverage-guided fuzzing with AFL [60] to improve the quality of a pre-existing test-suite. Their approach however only generated 275 minimized testcases from a first generation of 30,000 cases. In contrast, CUTECat’s concolic-based approach allows to generate cases exploring different execution paths, while reaching rare ambiguous or unhandled situations. Monat et al. [41] and de Almeida Borges et al. [17] focus on formalizing date-duration arithmetic and semantics of UTC time respectively, with the former also analyzing date-related ambiguities in legal computations. Our work currently only provides limited support for date operations, but could build upon these formalizations to define suitable symbolic encodings.

7 Conclusion and Future Work

Legal expert systems implementing computational laws are pervasive, and faithfully translating the law into code is error-prone. To alleviate this issue, we proposed CUTECat, a concolic execution tool targeting computational law implementations in the Catala language. CUTECat relies on a novel concolic semantics for default terms, and is equipped with several optimizations improving its scalability and usability by lawyers.

CUTECat scales to Catala codebases implementing real-world French and US laws, generating hundreds of thousands of unique testcases in less than seven hours of CPU time. We believe this paper takes a step towards the use of formal methods during legislative processes. Our future work includes tailoring CUTECat’s pipeline and interface for lawyers through interdisciplinary experimentation. We also plan to discuss additional usecases for CUTECat with law experts; namely, we believe CUTECat could be used to improve state-of-the-art computer-assisted tax teaching techniques [29].

Acknowledgments. We thank the anonymous reviewers for their constructive feedback and support of our work. We are grateful to Rohan Padhye for his suggestions around concolic testing for Catala, and to Patrick Baillot for feedback about earlier versions of this paper. We thank Liane Huttner & Sarah Lawsky for the interesting discussions around the properties this work targets, Louis Gesbert for his technical help around the Catala compiler, and Denis Merigoux for clarifications about the current implementation of default terms in Catala. We appreciated the many discussions and valuable feedback about this work we got from the whole Catala team.

References

- [1] Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* **51**(3), 50:1–50:39 (2018), <https://doi.org/10.1145/3182657>
- [2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: Cvc5: A versatile and industrial-strength SMT solver. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, *Lecture Notes in Computer Science*, vol. 13243, pp. 415–442, Springer (2022), https://doi.org/10.1007/978-3-030-99524-9_24
- [3] Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016), URL <https://www.SMT-LIB.org>
- [4] Barrett, C., Kroening, D., Melham, T.: Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories. Tech. Rep. 3, London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering (Jun 2014), URL <http://theory.stanford.edu/~barrett/pubs/BKM14.pdf>, knowledge Transfer Report
- [5] Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* **10**(6), 234–245 (1975), <https://doi.org/10.1145/390016.808445>
- [6] Brewka, G., Eiter, T.: Prioritizing default logic. In: *Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel*, pp. 27–45, Springer (2000)
- [7] Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2011), <https://doi.org/10.1145/1966445.1966463>
- [8] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008), URL <https://dl.acm.org/doi/10.5555/1855741.1855756>
- [9] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* (2008), <https://doi.org/10.1145/1455518.1455522>
- [10] Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. *Communications of the ACM* **56**(2), 82–90 (2013), <https://doi.org/10.1145/2408776.2408795>
- [11] Cassano, V., Fervari, R., Hoffmann, G., Areces, C., Castro, P.F.: A tableaux calculus for default intuitionistic logic. In: *Proceedings of the International*

- Conference on Automated Deduction (CADE) (2019), https://doi.org/10.1007/978-3-030-29436-6_10
- [12] Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I.: SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In: USENIX Security Symposium, pp. 2531–2548, USENIX Association (2022), URL <https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju>
 - [13] Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (2011), <https://doi.org/10.1145/1950365.1950396>
 - [14] Clarke, L.A.: A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* (3), 215–222 (1976), <https://doi.org/10.1109/TSE.1976.233817>
 - [15] Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability modulo Theories, Oxford, United Kingdom (Jul 2018), URL <https://inria.hal.science/hal-01960203>
 - [16] David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) (2016), <https://doi.org/10.1109/SANER.2016.43>
 - [17] de Almeida Borges, A., Bedmar, M.G., Rodríguez, J.J.C., Reyes, E.H., Buñuel, J.C., Joosten, J.J.: UTC time, formally verified. In: Proceedings of the International Conference on Certified Programs and Proofs (CPP), pp. 2–13, ACM (2024), <https://doi.org/10.1145/3580430>
 - [18] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008), https://doi.org/10.1007/978-3-540-78800-3_24
 - [19] Giantsios, A., Papaspyrou, N., Sagonas, K.: Concolic testing for functional languages. *Science of Computer Programming* **147**, 109–134 (2017), <https://doi.org/10.1016/j.scico.2017.04.008>
 - [20] Godefroid, P.: Compositional dynamic test generation. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL), pp. 47–54, POPL '07, Association for Computing Machinery, New York, NY, USA (2007), <https://doi.org/10.1145/1190216.1190226>
 - [21] Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), pp. 213–223, ACM (2005), <https://doi.org/10.1145/1065010.1065036>
 - [22] Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Sympos-

- sium (NDSS) (2008), URL <https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/>
- [23] Goutagny, P., Fromherz, A., Monat, R.: CUTEcat: Concolic Execution for Computational Law (Artifact) (Jan 2025), <https://doi.org/10.5281/zenodo.14677860>
 - [24] Howden, W.: Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering* **SE-3**(4), 266–278 (1977), <https://doi.org/10.1109/TSE.1977.231144>
 - [25] Howson, C.: *Logic with Trees: An Introduction to Symbolic Logic*. Routledge (2005)
 - [26] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* **37**(5), 649–678 (2011), <https://doi.org/10.1109/TSE.2010.62>
 - [27] King, J.C.: Symbolic execution and program testing. *Communications of the ACM* **19**(7), 385–394 (1976), <https://doi.org/10.1145/360248.360252>
 - [28] Lawsky, S.B.: A logic for statutes. *Fla. Tax Rev.* **21**, 60 (2017), <https://doi.org/10.2139/ssrn.3088206>
 - [29] Lawsky, S.B.: Teaching algorithms and algorithms for teaching (2021), <https://doi.org/10.5744/ftr.2021.2004>
 - [30] Linke, T., Schaub, T.: Lemma handling in default logic theorem provers. In: *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pp. 285–292, Springer (1995), https://doi.org/10.1007/3-540-60112-0_33
 - [31] Luckow, K., Dimjašević, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamarić, Z., Raman, V.: jDart: A dynamic symbolic analysis framework. In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2016), https://doi.org/10.1007/978-3-662-49674-9_26
 - [32] Majumdar, R., Sen, K.: Hybrid concolic testing. In: *Proceedings of the International Conference on Software Engineering (ICSE)* (2007), <https://doi.org/10.1109/ICSE.2007.41>
 - [33] Maranget, L.: Compiling pattern matching to good decision trees. In: *Proceedings of the ACM SIGPLAN Workshop on ML*, pp. 35–46, ACM (2008), <https://doi.org/10.1145/1411304.1411311>
 - [34] Marques, F., Fragoso Santos, J., Santos, N., Adão, P.: Concolic execution for WebAssembly. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, *Leibniz International Proceedings in Informatics (Lipics)*, vol. 222, pp. 11:1–11:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022), <https://doi.org/10.4230/LIPIcs.ECOOP.2022.11>
 - [35] Merigoux, D.: Experience report: Implementing a real-world, medium-sized program derived from a legislative specification. Tech. rep. (Jan 2023), URL <https://inria.hal.science/hal-03933574>
 - [36] Merigoux, D., Barnes, J., Botbol, V., Estep, S., Gerrior, M., Gesbert, L., Rolley, E., Salek, A., Scott, E., Török, E.: English tutorial for catala developers (2024), URL <https://catala-lang.org/en/examples/tutorial>

- [37] Merigoux, D., Chataing, N., Protzenko, J.: Catala: A programming language for the law. *Proceedings of the International Conference on Functional Programming (ICFP)* 5(ICFP), 77:1–77:29 (2021), <https://doi.org/10.1145/3473582>
- [38] Merigoux, D., Monat, R., Protzenko, J.: A modern compiler for the French tax code. In: *Proceedings of the International Conference on Compiler Construction (CC)*, pp. 71–82, CC 2021, ACM, New York, NY, USA (2021), <https://doi.org/10.1145/3446804.3446850>
- [39] Micinski, K., Fetter-Degges, J., Jeon, J., Foster, J.S., Clarkson, M.R.: Checking interaction-based declassification policies for android using symbolic execution. In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (2015), https://doi.org/10.1007/978-3-319-24177-7_26
- [40] Mockler, P.: The Phoenix Pay Problem: Working toward a Solution. Senate Canada (2018), URL <https://publications.gc.ca/pub?id=9.859900&sl=0>
- [41] Monat, R., Fromherz, A., Merigoux, D.: Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law. In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*, *Lecture Notes in Computer Science*, vol. 14577, pp. 421–450, Springer, Cham (2024), https://doi.org/10.1007/978-3-031-57267-8_16
- [42] Monin, J.: Louvois, le logiciel qui a mis l’armée à terre (2018), URL <https://www.radiofrance.fr/franceinter/podcasts/secrets-d-info/louvois-le-logiciel-qui-a-mis-l-armee-a-terre-8752880>
- [43] Nguyễn, P.C., Van Horn, D.: Relatively complete counterexamples for higher-order programs. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pp. 446–456, PLDI ’15, Association for Computing Machinery, New York, NY, USA (2015), <https://doi.org/10.1145/2737924.2737971>
- [44] Noller, Y., Kersten, R., Păsăreanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), <https://doi.org/10.1145/3213846.3213868>
- [45] Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic execution of Java bytecode. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 179–180, ASE ’10, Association for Computing Machinery, New York, NY, USA (2010), <https://doi.org/10.1145/1858996.1859035>
- [46] Poeplau, S., Francillon, A.: Symbolic execution with SymCC: Don’t interpret, compile! In: *Proceedings of the 29th USENIX Conference on Security Symposium*, pp. 181–198, SEC’20, USENIX Association, USA (2020), URL <https://dl.acm.org/doi/10.5555/3489212.3489223>
- [47] Redden, J., Brand, J., Sander, I., Warne, H., Grant, A., White, D.: Automating public services: Learning from cancelled systems. Cardiff, UK: Data Justice Lab, Cardiff University (2022),

- URL <https://carnegieuk.org/publication/automating-public-services-learning-from-cancelled-systems/>
- [48] Risch, V.: Analytic tableaux for default logics. *Journal of Applied Non-Classical Logics* **6**(1), 71–88 (1996), <https://doi.org/10.1080/11663081.1996.10510867>
 - [49] Schaub, T.: A new methodology for query answering in default logics via structure-oriented theorem proving. *Journal of Automated Reasoning* **15**, 95–165 (1995), <https://doi.org/10.1007/BF00881832>
 - [50] Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*, *Lecture Notes in Computer Science*, vol. 4144, pp. 419–423, Springer (2006), https://doi.org/10.1007/11817963_38
 - [51] Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: *ESEC/SIGSOFT FSE*, pp. 263–272, ACM (2005), <https://doi.org/10.1145/1081706.1081750>
 - [52] Sivaramakrishnan, KC., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. In: *Proceedings of the International Conference on Functional Programming (ICFP)* (2020), <https://doi.org/10.1145/3408995>
 - [53] Staats, M., Păsăreanu, C.: Parallel symbolic execution for structural test generation. In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* (2010), <https://doi.org/10.1145/1831708.1831732>
 - [54] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2016), <https://doi.org/10.14722/ndss.2016.23368>
 - [55] Tobin-Hochstadt, S., Van Horn, D.: Higher-order symbolic execution via contracts. In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 537–554, OOPSLA ’12, Association for Computing Machinery, New York, NY, USA (2012), <https://doi.org/10.1145/2384616.2384655>
 - [56] Vikram, V., Padhye, R., Sen, K.: Growing a Test Corpus with Bonsai Fuzzing. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 723–735, IEEE (May 2021), <https://doi.org/10.1109/ICSE43902.2021.00072>
 - [57] Wikipedia contributors: Code of Ur-Nammu — Wikipedia, the free encyclopedia (2024), URL https://en.wikipedia.org/w/index.php?title=Code_of_Ur-Nammu&oldid=1243860044
 - [58] Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: *USENIX Security Symposium*, pp. 745–761, USENIX Association (2018), URL <https://dl.acm.org/doi/10.5555/3277203.3277260>

- [59] Z3 Guide: Soft constraints (2024), URL <https://microsoft.github.io/z3guide/docs/optimization/softconstraints>
- [60] Zalewski, M.: American fuzzy lop. GitHub (Jul 2020), URL <https://github.com/google/AFL>

This work is licensed under a [Creative Commons “Attribution 4.0 International”](#) license.



Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





First-Person Choreographic Programming with Continuation-Passing Communications

Sung-Shik Jongmans^(✉)

University of Groningen, Groningen, The Netherlands

s.s.t.q.jongmans@rug.nl

Abstract. Choreographic programming (CP) is a method to implement distributed systems that ensures communication deadlock freedom by design. To use CP, though, the number of processes and the network among them must be known statically. Often, that information is known only dynamically. Thus, existing CP languages cannot be used to implement *process-parametric* distributed systems.

This paper introduces *first-person* choreographic programming (1CP) to support the implementation of process-parametric distributed systems while also ensuring communication deadlock freedom. We present both a design of 1CP (new calculus, formalised in Isabelle/HOL) and an implementation (new language and tooling, integrated in VS Code).

1 Introduction

1.1 Background: Choreographic Programming

Implementing distributed systems is hard. One of the challenges is to “prove”—broadly construed—the absence of *communication deadlocks* among message-passing processes. A communication deadlock is a form of misbehaviour that arises when all processes get stuck in cyclic communication dependencies.

Choreographic programming [3, 5, 39] is a top-down method to make the implementation of distributed systems easier. Fig. 1a visualises the idea:

1. First, construct a **global program** G (“choreography”). A global program defines the behaviour of all n processes in the distributed system, collectively.

Example 1. The *TwoBuyer* distributed system consists of processes *Buyer1* (**B1**), *Buyer2* (**B2**), and *Seller* (**S**). As introduced by Honda et al. [30]: “Buyer1 and Buyer2 wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller.” The following global program defines the behaviour of Buyer1, Buyer2, and Seller, collectively (excerpt):

```

 $G^{\text{TwoBuyer}} = \mathbf{B1}.\text{"foo"} \rightarrow \mathbf{S}.\text{title};$ 
 $\mathbf{S}.\text{quote}(\text{title}) \rightarrow [\mathbf{B1}, \mathbf{B2}].x; \quad // \text{ Seller computes the quote}$ 
 $\mathbf{B1}.(x / 3) \rightarrow \mathbf{B2}.y; \quad // \text{ Buyer1 contributes } 1/3$ 
 $\text{if } \mathbf{B2}.(x - y < x / 2) \dots \dots // \text{ Buyer2 contribs. at most } 1/2$ 

```

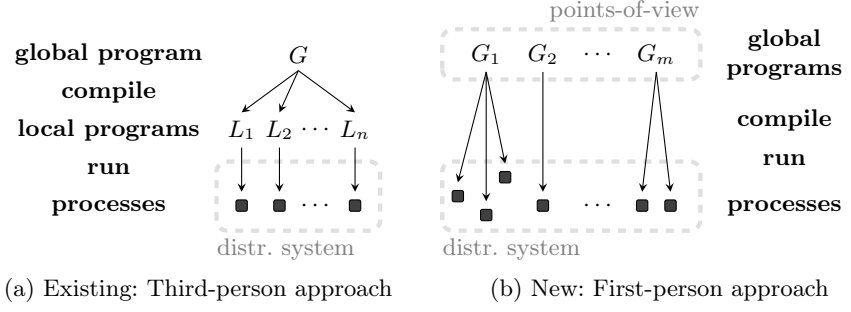


Fig. 1: Choreographic programming

Global program $p.E \rightarrow q.y$ defines a *communication* of (the value of) expression E , from the process named p , to the process named q , into variable y . Global programs **if** $p.E \ G_1 \ G_2$ and $G_1 ; G_2$ define *conditional choice*, at the process named p , and *sequencing*. We write $p.E \rightarrow [q_1 \dots, q_n].y$ as a shorthand for $p.E \rightarrow q_1.y ; \dots ; p.E \rightarrow q_n.y$. $\square$

2. Next, statically decompose the global program into **local programs** $L_1, L_2, \dots, L_n$, by **compiling** G separately for each of the n processes. A local program defines the behaviour of one process, individually.

Example 2. The following local programs, compiled from G^{TwoBuyer} , define the behaviour of Buyer1, Buyer2, and Seller, individually (excerpts):

$$\begin{aligned}
 L_{\mathbf{B1}}^{\text{TwoBuyer}} &= \mathbf{S}! \text{"foo"} ; \mathbf{S}?x ; \mathbf{B2}!(x/2) \\
 L_{\mathbf{B2}}^{\text{TwoBuyer}} &= \mathbf{S}?x ; \mathbf{B1}?y ; \text{if } (x - y < x/2) (\dots) (\dots) \\
 L_{\mathbf{S}}^{\text{TwoBuyer}} &= \mathbf{B1}?title ; \mathbf{B1}!\text{quote}(title) ; \mathbf{B2}!\text{quote}(title) ; (\dots)
 \end{aligned}$$

Local programs $q!E$ and $p?y$ define a *send* to the process named q and a *receive* from the process named p . $\square$

3. Last, dynamically (re)compose the local programs as **processes** into a distributed system by **running** them together, in parallel.

The main result of choreographic programming is that, if the compilation of G succeeds for each process, then the run of $L_1, L_2, \dots, L_n$ is free of communication deadlocks. That is, we enjoy *communication deadlock freedom by design*.

Early work on choreographic programming is due to Carbone et al. [3, 4] and Carbone–Montesi [5]; substantial progress has been made since. For instance, Montesi and Yoshida developed a theory of compositional choreographic programming [40]; Carbone et al. studied connections with linear logic [2, 6, 7]; Dalla Preda et al. combined choreographic programming with dynamic adaptation [45–47]; Cruz-Filipe et al. presented a technique to extract global programs from local programs [14]; Giallorenzo et al. studied a correspondence with multi-tier languages [26]. Furthermore, theoretical results are supported in practice by several tools, including Chor [5], Choral [25, 26], and HasChor [50].

1.2 Open Problem: Process-Parametric Distributed Systems

Central to the compilation of global programs is the concept of static *projection*. Formally, static projection is a function $\upharpoonright$ that consumes a global program G and a name p as input, and that produces a local program $G \upharpoonright p$ as output.

Essentially, to decompose global program G into local programs, the compiler statically projects G onto each name that occurs in G . In this way, *the number of names* that occur in G at compile-time fully determines *the number of processes* that occur in the distributed system at run-time. In many distributed systems, though, the number of processes that occur at run-time is known only dynamically, not statically. Such distributed systems cannot be implemented using existing choreographic programming languages.

Example 3. An *AnyBuyer* distributed system consists of processes *Buyer1* (**B1**), *Buyer2* (**B2**), ..., *Buyern* (**Bn**), and *Seller* (**S**). It generalises the *TwoBuyer* distributed system from two buyers to any number. Each buyer contributes a fraction of the quote; *Buyern* decides to accept/reject. The following global programs define the behaviour of all processes, collectively, for $n \in \{3, 4\}$ (excerpts):

$$\begin{array}{ll}
 G^{\text{ThreeBuyer}} = & G^{\text{FourBuyer}} = \\
 \mathbf{B1}.\text{"foo"} \rightarrow \mathbf{S}.\text{title}; & \mathbf{B1}.\text{"foo"} \rightarrow \mathbf{S}.\text{title}; \\
 \mathbf{S}.\text{quote}(\text{title}) \rightarrow [\mathbf{B1}, \mathbf{B2}, \mathbf{B3}].x; & \mathbf{S}.\text{quote}(\text{title}) \rightarrow [\mathbf{B1}, \mathbf{B2}, \mathbf{B3}, \mathbf{B4}].x; \\
 \mathbf{B1}.(x / f) \rightarrow \mathbf{B2}.y; & \mathbf{B1}.(x / f) \rightarrow \mathbf{B2}.y; \\
 \mathbf{B2}.(x / f + y) \rightarrow \mathbf{B3}.y; & \mathbf{B2}.(x / f + y) \rightarrow \mathbf{B3}.y; \\
 \text{if } \mathbf{B3}.(x - y < x / f) \dots \dots & \mathbf{B3}.(x / f + y) \rightarrow \mathbf{B4}.y; \\
 & \text{if } \mathbf{B4}.(x - y < x / f) \dots \dots
 \end{array}$$

Each buyer has its own variables x , y , and f (initialised when the run begins).

Thus, for each n , a separate global program needs to be written: no existing choreographic programming language supports *parametrisation* in n . However, n is typically known only dynamically. As a result, realistically, *AnyBuyer* cannot be implemented using choreographic programming. $\square$

Generally, existing choreographic programming languages require not only the number of processes to be known statically (always one per name), but also *the network* among them (always fully connected). These constraints seriously limit the applicability of choreographic programming. Thus, the **open problem** addressed in this paper is how to support **process-parametric distributed systems**, in which such information is known only dynamically. Related work suggests that this is difficult: in *multiparty session typing* (MPST) [30], in which static projection is used to decompose *global types* into *local types*, extensions for process parametricity lose decidability [51], soundness [41], or expressiveness [8].

1.3 This Paper

Whereas the decomposition of global types in MPST *must* happen statically (i.e., local types are used at compile-time), the decomposition of global programs in

choreographic programming *may*—in principle—happen dynamically (i.e., local programs are used only at run-time). That is, in contrast to MPST, there is no fundamental reason in choreographic programming to insist on *static* projection. This insight paves the way to a completely different approach to choreographic programming based on *dynamic* projection. Exploring this approach is the aim of this paper. To summarise the contributions:

- **What:** A novel approach, called *first-person choreographic programming*
- **How:** A new choreographic primitive, called *continuation-passing communication*, that enables *dynamic projection* (at run-time) of choreographies that are still *statically well-typed* (at compile-time)
- **Why:** To implement *process-parametric distributed systems* using choreographic programming, including *communication deadlock freedom by design*

Concretely, we present both a design of first-person choreographic programming (new calculus, formalised in Isabelle/HOL) and an implementation (new language and tooling, integrated in VS Code). [Sect. 2](#) presents an overview. [Sect. 3](#) presents the design. [Sect. 4](#) presents the implementation (including type checking and interpretation; without code generation). The paper is concluded in [Sect. 5](#).

2 Overview

2.1 Design: A Tour of First-Person Choreographic Programming

In existing choreographic programming languages, global programs are written from the *third-person point-of-view of the programmer*. For instance, $\mathbf{A}.5 \rightarrow \mathbf{B}.x$ means “a communication from Alice to Bob” from the programmer’s point-of-view. At heart, the key novelty of this paper is to shift the narrative perspective in global programs to the *first-person points-of-view of the processes*. For instance, $5 \rightarrow \mathbf{B}.x$ means “a communication from *me* to Bob” from Alice’s point-of-view. This shift in perspective is instrumental for us to support process-parametricity.

We call the existing approach *third-person choreographic programming* (3CP) and our new approach *first-person choreographic programming* (1CP). [Fig. 1b](#) visualises the idea. We clarify it below and introduce our calculus along the way.

1. First, construct global programs $G_1, G_2, \dots, G_m$. Like in 3CP, each global program in 1CP defines the behaviour of all processes, collectively. However:
 - In 3CP, a *single* global program reflects *multiple* points-of-view, each taken by *one* process.
 - In 1CP, in contrast, *multiple* global programs each reflect a *single* point-of-view, taken by *one-or-more* processes (process parametricity).

Example 4. To clarify the terminology and difference between 3CP and 1CP:

- In 3CP, TwoBuyer consists of three processes (Buyer1, Buyer2, Seller), and three points-of-view (Buyer1, Buyer2, Seller; we overload the process names when the correspondence is one-to-one), reflected by G^{TwoBuyer} .

- In 1CP, AnyBuyer consists of $n+1$ processes (Buyer1, ..., Buyer n , Seller), but three points-of-view (Buyer1, Buyer ≥ 2 , Seller), reflected by three global programs. Thus, a single point-of-view (Buyer ≥ 2) is taken by more than one process (Buyer2, ..., Buyer n). We note that Buyer1 has a different point-of-view than Buyer i , for any $i \geq 2$, as Buyer1 is initially *active* (sends a message), whereas Buyer i is initially *reactive* (awaits a message). In our calculus, the following global programs reflect this:

$$\begin{aligned}
 G_{\mathbf{B1}}^{\text{AnyBuyer}} &= \dots \text{ (Exmp. 5)} && // \text{ Point-of-view: Buyer1} \\
 G_{\mathbf{Bi}}^{\text{AnyBuyer}} &= \text{skip} && // \text{ Point-of-view: Buyer} \geq 2 \\
 G_{\mathbf{S}}^{\text{AnyBuyer}} &= \text{skip} && // \text{ Point-of-view: Seller}
 \end{aligned}$$

Global program **skip** defines a *no-op*. Thus, $G_{\mathbf{Bi}}^{\text{AnyBuyer}}$ and $G_{\mathbf{S}}^{\text{AnyBuyer}}$ define the absence of initial behaviour. $\square$

In more detail, each global program is written in terms of *continuation-passing communications* and *choreographic procedures*.

Example 5. To demonstrate continuation-passing communications, in our calculus, the following global program reflects the point-of-view of Buyer1:

$$\begin{aligned}
 G_{\mathbf{B1}}^{\text{AnyBuyer}} = & \\
 & \text{"foo"} \rightarrow \mathbf{S}.\text{title} \triangleright && // \text{ Active: Buyer1} \\
 & \text{quote}(\text{title}) \rightarrow \mathbf{B1}.\mathbf{x} \triangleright && // \text{ Active: Seller} \\
 & (\mathbf{x} / \mathbf{f}) \rightarrow \mathbf{neigh}.\mathbf{y} \triangleright && // \text{ Active: Buyer1} \\
 & \text{contrib}; && // \text{ Active: Buyer2} \\
 & \text{foreach}_{\mathbf{Bi}} \text{ neigh} \in \mathbf{neighs} \text{ do} && // \text{ Active: Seller} \\
 & \quad \text{quote}(\text{title}) \rightarrow \mathbf{neigh}.\mathbf{x} \triangleright && // \text{ Active: Seller} \\
 & \quad \text{contrib} && // \text{ Active: Buyer}_i, \text{ for any } i \geq 2
 \end{aligned}$$

Global program $E \rightarrow q.y \triangleright G$ defines a *continuation-passing communication*

- of (the value of) expression E , from an implicit *active process*,
- to the explicit *reactive process* named q , into variable y ,
- followed by the asynchronous execution, by q , of global program G , written from q 's point-of-view.

Thus, as the first active process, Buyer1 passes "foo" and a continuation—written from Seller's point-of-view—to Seller. When Seller receives, as the next active process, it passes `quote(title)` and a continuation to Buyer1.

- When Buyer1 receives, as one of the now-active processes, it passes $\mathbf{x} / \mathbf{f}$ and a continuation to **neigh** (a reference to Buyer2). When Buyer2 receives, it calls choreographic procedure **contrib**.
- In parallel, Seller—still active—passes `quote(title)` and a continuation to each **neigh** in **neighs** (a list of references to Buyer2, ..., Buyer n).

This example demonstrates that, due to the asynchronous execution of continuations, multiple processes can become active in parallel. $\square$

Intuitively, each occurrence of “ $\triangleright$ ” signifies a lexical context switch in which the previous active process passes a message and a continuation to the next active process.¹ In this way, continuation passing is a form of *process-driven* dynamic projection: the processes at run-time tell each other how to behave (from their first-person points-of-view), instead of a choreographic programming tool at compile-time (from its third-person point-of-view).

Example 6. To demonstrate choreographic procedures, in our calculus, the following *choreographic libraries* map choreographic procedure *names* to *bodies*, for each of the three points-of-view in AnyBuyer:

$$\begin{aligned}\Psi_{\mathbf{B1}}^{\text{AnyBuyer}} &= \emptyset && // \text{Point-of-view: Buyer1} \\ \Psi_{\mathbf{B}i}^{\text{AnyBuyer}} &= \{\text{contrib} \mapsto G_{\text{contrib}}^{\text{AnyBuyer}}\} && // \text{Point-of-view: Buyer} \geq 2 \\ \Psi_{\mathbf{S}}^{\text{AnyBuyer}} &= \emptyset && // \text{Point-of-view: Seller}\end{aligned}$$

Thus, Buyer1 and Seller do not have choreographic procedures, while Buyer i , for any $i \geq 2$, has **contrib**. The following global program defines its body:

$$\begin{aligned}G_{\text{contrib}}^{\text{AnyBuyer}} &= \text{if } (\mathbf{x} \neq 0 \ \&\& \ \mathbf{y} \neq 0) \\ &\quad \text{if last} \\ &\quad \quad \text{if } (\mathbf{x} - \mathbf{y} < \mathbf{x} / \mathbf{f}) \ (\dots) \ (\dots) \\ &\quad \quad (\mathbf{x} / \mathbf{f} + \mathbf{y}) \rightarrow \text{neigh.y} \triangleright \text{contrib}\end{aligned}$$

Thus, each time Buyer i calls **contrib**, it checks if $\mathbf{x}$ (quote) and $\mathbf{y}$ (contribution up to Buyer $i-1$) are already set; 0 indicates unset. If so, and if **last** (flag to indicate whether or not Buyer i is the n -th buyer) is true, then Buyer i decides to accept/reject the quote; otherwise, using $\mathbf{f}$ (denominator of the fraction that Buyer i can contribute), Buyer i passes $\mathbf{x} / \mathbf{f} + \mathbf{y}$ (total contribution up to Buyer i) and a continuation to **neigh** (reference to Buyer $i+1$).

Putting the pieces in this example and in [Exmp. 5](#) together, Buyer i calls **contrib** twice: once in a continuation passed to it by Seller (and the incoming message sets $\mathbf{x}$), and once in a continuation passed to it by Buyer $i-1$ (and the incoming message sets $\mathbf{y}$), but in any order (due to asynchrony). $\square$

2. Next, **at compile-time**, statically analyse each global program, including the bodies of choreographic procedures, by means of *type checking*. Our main result ([Thm. 1](#)) is that if each global program is well-typed, then any *properly initialised* run of those global programs is free of communication deadlocks. Proper initialisation means that the initial values of variables are well-typed; this involves a lightweight check right at the beginning of a run. Thus, just as third-person choreographic programming, first-person choreographic programming guarantees communication deadlock freedom by design.

¹ Passing code around in this way is a programming abstraction. It can be “compiled away” and avoided at run-time; see the end of [Sect. 4.2](#).

Example 7. Intuitively, to ensure that none of the processes in AnyBuyer get stuck, they must have the following variables and choreographic procedures:

- Buyer1: x, f (numbers) and **neigh** (process reference);
- Buyer2, ..., Buyer n : x, y, f (numbers), **last** (boolean), **neigh** (process reference), and **contrib** (choreographic procedure);
- Seller: **title** (string) and **neighs** (list of process references).

In our calculus, the following *data/choreographic type environments* capture these assumptions, for each of the three points-of-view:

$$\begin{aligned}
 \Gamma_{\mathbf{B1}}^{\text{AnyBuyer}} &= \{x \mapsto \text{Nat}, f \mapsto \text{Nat}, \text{neigh} \mapsto \text{Proc}\langle \mathbf{B1} \rangle\} \\
 \Gamma_{\mathbf{B1}}^{\text{AnyBuyer}} &= \{x \mapsto \text{Nat}, y \mapsto \text{Nat}, f \mapsto \text{Nat}, \text{last} \mapsto \text{Bool}, \text{neigh} \mapsto \text{Proc}\langle \mathbf{B1} \rangle\} \\
 \Gamma_{\mathbf{S}}^{\text{AnyBuyer}} &= \{\text{title} \mapsto \text{Str}, \text{neighs} \mapsto \text{List}\langle \text{Proc}\langle \mathbf{B1} \rangle \rangle\} \\
 \Delta_{\mathbf{B1}}^{\text{AnyBuyer}} &= \emptyset \quad \Delta_{\mathbf{B1}}^{\text{AnyBuyer}} = \{\text{contrib} \mapsto \text{Chor}\langle \mathbf{B1} \rangle\} \quad \Delta_{\mathbf{S}}^{\text{AnyBuyer}} = \emptyset
 \end{aligned}$$

Using these environments, we can statically prove that $G_{\mathbf{B1}}^{\text{AnyBuyer}}$, $G_{\mathbf{B1}}^{\text{AnyBuyer}}$, $G_{\mathbf{S}}^{\text{AnyBuyer}}$, and $G_{\text{contrib}}^{\text{AnyBuyer}}$ (all defined in previous examples) are well-typed; we do not need to know n to make this judgment. Thus, any properly initialised run—for any n —is free of communication deadlocks.

We note that the typing rules do not guarantee functional correctness. For instance, if both Buyer2 and Buyer3 have a reference to Buyer4 in **neigh**, then the system is deadlock-free (well-typed) but functionally incorrect. $\square$

3. Last, **at run-time**, dynamically create a *configuration* for each process, and compose those configurations into a distributed systems by running them together, in parallel. If the point-of-view of a process is p , then its initial configuration is of the shape (Φ, Ψ, G) , where:
 - Φ defines the initial values of the variables of the process;
 - Ψ defines the choreographic library from point-of-view p ;
 - G defines the behaviour of all processes, collectively, from point-of-view p .
 Thus, all processes that have the same point-of-view, have the same choreographic library and the same global program in their initial configurations. To be able to reference different processes that have the same point-of-view, we use *indexed names* of the shape $p[i]$, where p is a point-of-view, and i is an index. We write just p as a shorthand for $p[0]$.

Example 8. The following configurations initialise AnyBuyer for $n = 4$:

$$(\Phi_{p[i]}, \Psi_p^{\text{AnyBuyer}}, G_p^{\text{AnyBuyer}}) \quad \text{for } p[i] \in \{\mathbf{B1}, \mathbf{B1}[2], \mathbf{B1}[3], \mathbf{B1}[4], \mathbf{S}\}$$

$$\begin{aligned}
 \text{where } \Phi_{\mathbf{B1}} &= \{\text{neigh} \mapsto \mathbf{B1}[2], x \mapsto 0, f \mapsto 3\} \\
 \Phi_{\mathbf{B1}[2]} &= \{\text{neigh} \mapsto \mathbf{B1}[3], x \mapsto 0, y \mapsto 0, f \mapsto 5, \text{last} \mapsto \text{false}\} \\
 \Phi_{\mathbf{B1}[3]} &= \{\text{neigh} \mapsto \mathbf{B1}[4], x \mapsto 0, y \mapsto 0, f \mapsto 8, \text{last} \mapsto \text{false}\} \\
 \Phi_{\mathbf{B1}[4]} &= \{\text{neigh} \mapsto \mathbf{B1}[4], x \mapsto 0, y \mapsto 0, f \mapsto 2, \text{last} \mapsto \text{true}\} \\
 \Phi_{\mathbf{S}} &= \{\text{neighs} \mapsto [\mathbf{B1}[2], \mathbf{B1}[3], \mathbf{B1}[4]], \text{title} \mapsto ""\}
 \end{aligned}$$

Each $\Phi_{p[i]}$ is well-typed in $\Gamma_p^{\text{AnyBuyer}}$; this is a near-trivial check. $\square$

Example 9. To demonstrate a run, we show the first few *reductions* of the configurations of Buyer1 and Seller in our calculus. First, the following configurations are initial:

$$\begin{aligned} \mathbf{B1} &\mapsto (\Phi_{\mathbf{B1}}, \Psi_{\mathbf{B1}}, \text{"foo"} \rightarrow \mathbf{S.title} \triangleright ((\text{quote}(\text{title}) \rightarrow \mathbf{B1.x} \triangleright G_1); G_2)) \\ \mathbf{S} &\mapsto (\Phi_{\mathbf{S}}, \Psi_{\mathbf{S}}, \text{skip}) \end{aligned}$$

Next, Buyer1 passes "foo" and a continuation to Seller:

$$\begin{aligned} \mathbf{B1} &\mapsto (\Phi_{\mathbf{B1}}, \Psi_{\mathbf{B1}}, \text{skip}) \\ \mathbf{S} &\mapsto (\Phi_{\mathbf{S}}, \Psi_{\mathbf{S}}, \text{title} := \text{"foo"}; (\text{quote}(\text{title}) \rightarrow \mathbf{B1.x} \triangleright G_1); G_2) \end{aligned}$$

Global program $x := E$ defines the assignment of (the value of) E , to variable x . Thus, as part of the communication from Buyer1 to Seller, the assignment of the incoming message and the execution of the continuation are explicitly scheduled to be executed by Seller, using sequencing.

Next, Seller executes the assignment. Let $\Phi'_{\mathbf{S}} = \Phi_{\mathbf{S}}\{\text{title} \mapsto \text{"foo"}\}$:

$$\mathbf{B1} \mapsto (\Phi_{\mathbf{B1}}, \Psi_{\mathbf{B1}}, \text{skip}) \quad \mathbf{S} \mapsto (\Phi'_{\mathbf{S}}, \Psi_{\mathbf{S}}, (\text{quote}(\text{title}) \rightarrow \mathbf{B1.x} \triangleright G_1); G_2)$$

Next, Seller passes the value of $\text{quote}(\text{title})$ (i.e., this expression is eagerly evaluated at Seller instead of lazily at Buyer1) and a continuation to Buyer1. Let $\Phi'_{\mathbf{S}}[\text{quote}(\text{title})] = 123$ (the evaluation of $\text{quote}(\text{title})$ in $\Phi'_{\mathbf{S}}$):

$$\mathbf{B1} \mapsto (\Phi_{\mathbf{B1}}, \Psi_{\mathbf{B1}}, x := 123; G_1) \quad \mathbf{S} \mapsto (\Phi'_{\mathbf{S}}, \Psi_{\mathbf{S}}, G_2)$$

Next, Buyer1 executes the assignment or, in parallel, Seller asynchronously executes G_2 . And so on. $\square$

2.2 Implementation: 1CPLT by Example

About To explore how to possibly transfer the calculus from theory to practice, we initiated the *1CPLT* project. It consists of a prototype language, along with proof-of-concept tooling (including modern editor support), to apply first-person choreographic programming “for real”. It is built as an extension to *VS Code*.²

1CPLT includes a parser (implements the syntax of the calculus), a type checker (static semantics), and an interpreter (dynamic semantics). The examples below demonstrate 1CPLT implementations of basic distributed systems that are not supported by existing choreographic programming languages.

M-out-of-N First, we demonstrate the structure of the 1CPLT language.

Example 10. An *M-out-of-N* distributed system consists of processes *Producer*₁, ..., *Producer*_{*n*}, and *Consumer*. A self-reference is communicated from each producer to the consumer, unordered. The first *m* references received by the consumer, from any subset of producers, are stored; the last *n*−*m* references are

```

1  global Prod(ok: boolean)
2  {
3    main:
4      self.self → Cons.p |>
5      if
6        Cons.(ps.length < m)
7      then
8        Cons.ps := cons(p, ps) ;
9        Cons.true → p.ok
10     else
11       Cons.false → p.ok
12   }
13
14  global Cons(p: Prod, ps: Prod[], m: number)
15
16  Simulate all | Simulate Prod[1]
17  process Prod[1](false)
18  Simulate all | Simulate Prod[2]
19  process Prod[2](false)
20  Simulate all | Simulate Prod[3]
21  process Prod[3](false)
22  Simulate all | Simulate Cons
23  process Cons(Prod[1], [], 2)
24  // ~~~~~
25  // Dummy initial value

```

(a) Well-typed

```

3    main:
4      self.self → Cons.p |>
5      if
6        Cons.(ps.length < m)
7      then
8        Cons.ps := cons(p, ps) ;
9        Cons.true → p.ok
10     else
11       Cons.false → p.ok
12   }

```

(b) Ill-typed: Unexpected type

```

3    main:
4      self.self → Cons.p |>
5      if
6        Cons.(ps.length < m)
7      then
8        Cons.ps := cons(p, ps) ;
9        Cons.true → p.ok |> p.doMore
10     else
11       Cons.false → p.ok
12   }

```

(c) Ill-typed: Unexpected procedure

Fig. 2: M-out-of-N in 1CPLT

lost. The consumer tells each producer whether or not its reference was stored. There are two points-of-view: that of any producer and that of the consumer.

Fig. 2a shows a 1CPLT implementation of M-out-of-N.

Lines 1–14 contain two `global` blocks, each of which pertains to one point-of-view. In general, each `global` block defines:

- the name of a point-of-view (e.g., `Prod`);
- the variables of any process that has that point-of-view (e.g., `ok`);
- a distinguished choreographic procedure `main` whose body is the initial global program of any process that has that point-of-view (when `main` is absent, the initial global program is assumed to be `skip`);
- additional choreographic procedures (none in this example) that constitute the choreographic library of any process that has that point-of-view.

Lines 16–19 contain four `process` statements: each line defines an initial configuration of a process in terms of the name of its point-of-view (e.g., `Prod`), its index (e.g., 1; when absent, it is assumed to be 0), and its initial values of variables.

A key point is that **only global blocks are needed for static type checking, while process statements are needed only for dynamic execution.**

We note that the consumer also uses each received reference `p` to reply. $\square$

² <https://github.com/microsoft/vscode>

Next, we demonstrate two forms of modern editor support offered by 1CPLT to improve the first-person choreographic programming experience.

Example 11. Fig. 2 shows that 1CPLT leverages two special features of VS Code:

- **Inlay hints** provide the programmer contextual information about the code, displayed in grey font wherever relevant. In 1CPLT, inlay hints inform the programmer about which process is active by prefixing code fragments with process references, *in the style of third-person choreographic programming*. For instance, “`self.`” on line 4 and “`Cons.`” on lines 6, 8–9, and 11, are inlay hints: they are *not* part of the code fragments but inserted by 1CPLT’s inlay hint generator. Inlay hints are intended to improve code comprehension.
- **Code lenses** allow the programmer to trigger contextual actions, displayed as clickable text between lines of code (e.g., lines 16–19). In 1CPLT, “Simulate ...” uses the 1CPLT interpreter to run a simulation either of the full distributed system or of one particular process. $\square$

Last, we demonstrate 1CPLT implementations with errors.

Example 12. Fig. 2b and Fig. 2c show examples of bugs caught by 1CPLT:

- In Fig. 2b, an unexpected message is passed from the consumer to a producer.
- In Fig. 2c, a continuation with an unexpected choreographic procedure call is passed from the consumer to a producer.

In both cases, the consumer burdens a producer with input that it cannot handle, which causes that producer to fail. The 1CPLT type checker, which implements the typing rules of the calculus, detects this. $\square$

Even when m and n are fixed, most existing choreographic programming languages cannot be used to implement M-out-of-N. This is because these languages have limited support for non-determinism, which is crucial in M-out-of-N. That is, even when m and n are known statically, the order in which the consumer receives from the producers—and *how to reply*—is known only dynamically, which is problematic. In contrast, the event-driven nature of this paper’s approach supports non-deterministic receives. Another notable exception is the recent work by Plyukhin et al. [43], which offers special support for out-of-order execution.

Chang–Roberts In the remainder of this section, we further demonstrate the expressiveness of 1CPLT (and, in turn, of the new calculus) by considering two classical *distributed algorithms*. First, we consider a distributed algorithm for *leader election*. This is the problem of computing a unique leader among the processes in a distributed system. The *Chang–Roberts* algorithm solves the problem for *non-anonymous* processes in a directed ring network [11].

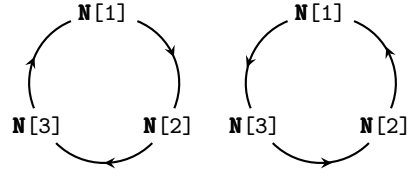
Example 13. A *Chang–Roberts* distributed system consists of processes *Node1*, ..., *Noden*. Each node has a unique numeric identifier. The nodes elect the node

```

1  global N(neigh: N, x, y: number, z: boolean)
2  {
3    main:
4      self.x -> neigh.y |> neigh.forward
5
6    forward:
7      if
8        self.x < y
9      then
10       self.y -> neigh.y |> neigh.forward
11      else
12       self.z := x == y // Leader?
13  }
14
15  process N[1](N[2], 5, 0, false)
16  process N[2](N[3], 6, 0, false)
17  process N[3](N[1], 7, 0, false)

```

(a) Clockwise



(b) Directions of the ring

```

13  }
14
15  process N[1](N[3], 5, 0, false)
16  process N[2](N[1], 6, 0, false)
17  process N[3](N[2], 7, 0, false)

```

(c) Anticlockwise

Fig. 3: Chang–Roberts

with the greatest identifier among them as the leader. There is only one point-of-view: that of any node.

Fig. 3a shows a 1CPLT implementation of Chang–Roberts. As there is only one point-of-view, there is only one `global` block. Each node has four variables: a reference to its neighbour (`neigh`), its own identifier (`x`), the greatest identifier it has received so far (`y`), and a flag to indicate whether or not it is elected (`z`).

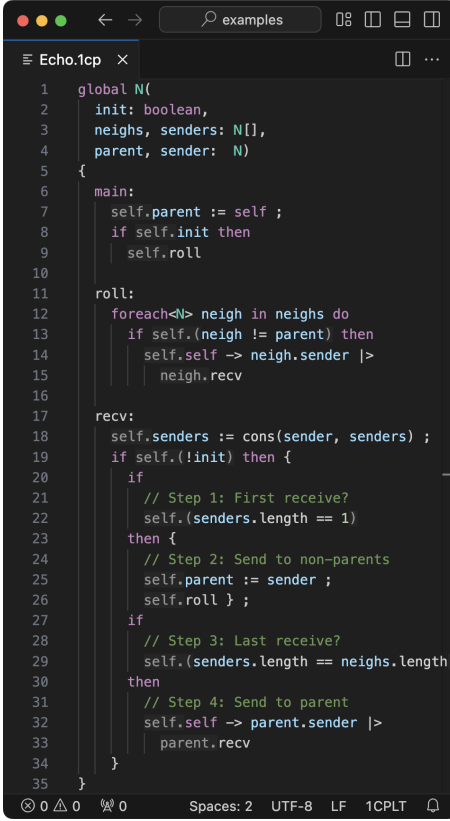
First, as defined by choreographic procedure `main`, the identifier of each node is passed to that node’s neighbour (i.e., each node is initially active). Next, as defined by choreographic procedure `forward`, if the neighbour’s own identifier is less than the incoming identifier, then the neighbour forwards the incoming identifier to the neighbour’s neighbour. And so on. In this way, only the greatest identifier is forwarded by all nodes and makes a full pass along the ring. When a node receives its own identifier, it declares itself the leader. $\square$

Next, we demonstrate that the network among the processes can be initialised at run-time, through the process references in their states.

Example 14. Fig. 3a and Fig. 3c show, on lines 15–17, definitions of two alternative sets of initial configurations for three nodes: a clockwise-directed ring and an anticlockwise-directed ring. That is, statically, it is known *that* each node has a neighbour, but only dynamically, it is known *which* one. $\square$

Even when the number of nodes and the neighbours of all nodes were fixed, *but not the identifiers*,³ existing choreographic programming languages cannot

³ If the identifiers were fixed, the process with the greatest identifier would be known already at compile-time, so there would be no need to elect a leader at run-time.

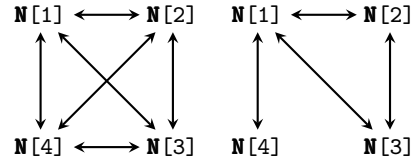


```

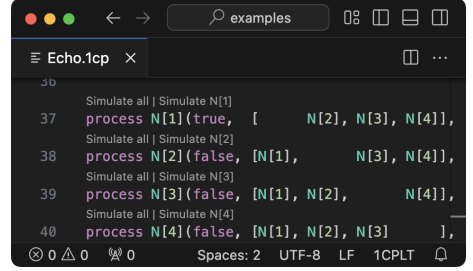
1  global N(
2    init: boolean,
3    neighs, senders: N[],
4    parent, sender: N)
5  {
6    main:
7      self.parent := self ;
8      if self.init then
9        self.roll
10
11  roll:
12    foreach<N> neigh in neighs do
13      if self.(neigh != parent) then
14        self.self -> neigh.sender |>
15        neigh.recv
16
17  recv:
18    self.senders := cons(sender, senders) ;
19    if self.(!init) then {
20      if
21        // Step 1: First receive?
22        self.(senders.length == 1)
23      then {
24        // Step 2: Send to non-parents
25        self.parent := sender ;
26        self.roll ;
27      }
28      if
29        // Step 3: Last receive?
30        self.(senders.length == neighs.length)
31      then
32        // Step 4: Send to parent
33        self.self -> parent.sender |>
34        parent.recv
35    }
36  }

```

(a) Global program



(b) Two possible topologies

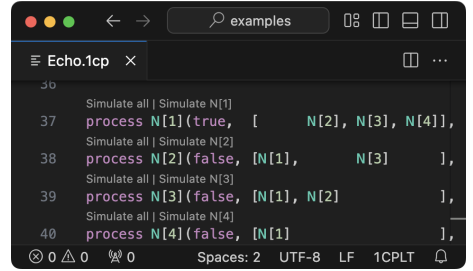


```

37  Simulate all | Simulate N[1]
38  process N[1](true, [ N[2], N[3], N[4] ],
39  Simulate all | Simulate N[2]
40  process N[2](false, [ N[1], N[3], N[4] ],
41  Simulate all | Simulate N[3]
42  process N[3](false, [ N[1], N[2], N[4] ],
43  Simulate all | Simulate N[4]
44  process N[4](false, [ N[1], N[2], N[3] ],

```

(c) Processes for left topology



```

37  Simulate all | Simulate N[1]
38  process N[1](true, [ N[2], N[3], N[4] ],
39  Simulate all | Simulate N[2]
40  process N[2](false, [ N[1], N[3] ],
41  Simulate all | Simulate N[3]
42  process N[3](false, [ N[1], N[2] ],
43  Simulate all | Simulate N[4]
44  process N[4](false, [ N[1] ],

```

(d) Processes for right topology

Fig. 4: Echo

be used to implement Chang–Roberts. This is because the total number of communications during a run (message complexity) depends on the assignment of identifiers to nodes (key property of Chang–Roberts). For instance, if the identifiers happen to be assigned in ascending order in the direction of the ring, then $2 \cdot n - 1$ communications are needed. In contrast, if the identifiers happen to be assigned in descending order, then $\frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ communications are needed. Existing choreographic programming language require at least n^2 communications.

Echo Next, we consider a distributed algorithm for *waves*. This is the problem of scattering/gathering information to/from all processes. The *Echo* algorithm solves the problem for undirected networks of *any topology* [49].

Example 15. An *Echo* distributed system consists of processes *Node1*, ..., *Noden*. One of the nodes acts as the *initiator* that triggers the wave. Each node except the initiator goes through the following steps:

1. Receive a message from any neighbour; it becomes the node's *parent*.
2. Send a message to each non-parent neighbour.
3. Receive a message back from each non-parent neighbour.
4. Send a message to the parent.

The initiator skips steps 1 and 4, while it sends/receives a message to/from all neighbours in steps 2 and 3. When step 3 of the initiator ends, it is guaranteed that each node has participated (and has been given the opportunity to scatter/gather information accordingly). Because the initiator and the non-initiators are very similar, we consider there to be only one point-of-view: that of any node. In general, there is some flexibility in determining the granularity at which points-of-view are identified for a given distributed system; this is a design decision.

Fig. 4a shows a 1CPLT implementation of Echo. Each node has five variables: a flag to indicate “initiator-hood” (**init**), a constant list of references to all neighbours (**neighs**), a list of references to all neighbours from which it has received (**senders**), a reference to the parent neighbour (**parent**), and a reference to the latest neighbour from which it has received (**sender**).

First, as defined by choreographic procedure **main**, each node checks if it is the initiator (i.e., each node is initially active, but only one of them triggers the wave). Next, as defined by choreographic procedure **roll**, the initiator sends a message and a continuation to each neighbour (i.e., **parent** has **self** as dummy initial value, which is not in **neighs**). Next, as defined by choreographic procedure **recv**, each neighbour of the initiator:

- updates **senders**;
- if it is the first receive, then it sets its parent and calls **roll** to send a message and a continuation to each neighbour (executed asynchronously by them);
- if it is the last receive, then it sends a message and a continuation back to its parent. □

Example 16. Fig. 4c and Fig. 4d show definitions of two alternative sets of initial configurations for four nodes, each of which has a different topology. The figure highlights that we need to write and statically analyse only one (collection of) global program(s) that can subsequently be initialised for any network. □

3 Design: Calculus of First-Person Choreographies

3.1 Organisation of the Section

Having introduced the main ingredients of the new calculus in Sect. 2.1, we present the details in this section. In three separate subsections, we define the syntax of the calculus (Sect. 3.2), its static semantics (Sect. 3.3), and its dynamic semantics (Sect. 3.4). Our presentation in these sections is layered: the first layer concerns “data to communicate”, while on top of that, the second layer concerns “choreographies to communicate data”. A final subsection states our main result (Sect. 3.5): communication deadlock freedom by design. All definitions, as well as the main result, are formalised in Isabelle/HOL; see also the artifact [35].

3.2 Syntax

Data We define the following syntax for data:

- Let $\mathbb{X} = \{x, y, z, \dots\}$ denote the set of *variables*, ranged over by x, y, z .
- Let $\mathbb{P}$ denote the set of *point-of-view names*, ranged over by p, q, r .
- Let $\mathbb{T}$ denote the set of *data types*, ranged over by T :

$$T ::= \text{Proc}\langle p \rangle \mid \text{Bool} \mid \text{Nat} \mid \text{List}\langle T \rangle \mid \dots$$

We note that many more data types may be available, but this is orthogonal to the contributions of this paper, so they are omitted here.

- Let $\mathbb{V}$ denote the set of *values*, ranged over by V :

$$V_{\text{Nat}} ::= 0 \mid 1 \mid 2 \mid \dots$$

$$V ::= p[V_{\text{Nat}}] \mid \text{true} \mid \text{false} \mid V_{\text{Nat}} \mid [V_1, \dots, V_n]\langle T \rangle \mid \dots$$

Value $p[V_{\text{Nat}}]$ —an “indexed point-of-view name”—defines a *process reference*.

Value $[V_1, \dots, V_n]\langle T \rangle$ defines a *list* of length n with elements of data type T .

- Let $\mathbb{E}$ denote the set of *expressions*, ranged over by E :

$$E ::= x \mid V \mid !E \mid E_1 \&\& E_2 \mid E_1 \parallel E_2 \mid \dots$$

- Let $\mathbb{X} \rightarrow \mathbb{V}$ denote the set of *process states* (partial functions from variables to values), ranged over by Φ .

Choreographies We define the following syntax for choreographies:

- Let $\mathbb{K}$ denote the set of *choreographic procedure names*, ranged over by k .
- Let $\mathbb{C} = \{\text{Chor}\langle p \rangle \mid p \in \mathbb{P}\}$ denote the set of *choreographic types*.
- Let $\mathbb{G}$ denote the set of global programs, ranged over by G :

$$G ::= k \mid \text{skip} \mid x := E \mid E_1 \rightarrow E_2.y \triangleright G \mid G_1 ; G_2 \mid \text{if } E \text{ } G_1 \text{ } G_2 \mid \text{while } E \text{ } G$$

Global program k defines a *call* of choreographic procedure k . Global program **skip** defines a *no-op*. Global program $x := E$ defines an *assignment* of (the value of) E , to variable x . Global program $E_1 \rightarrow E_2.y \triangleright G$ defines a *continuation-passing communication* of (the value of) E_1 , to (the value of) E_2 —“from an implicit active process to the explicit reactive process”—into variable y , followed by the asynchronous execution of G . Thus, E_1 is evaluated to the value to communicate, while E_2 is evaluated to (a reference to) the process to communicate with. Global programs $G_1 ; G_2$, **if** $E \text{ } G_1 \text{ } G_2$, and **while** $E \text{ } G$ define sequencing, conditional choice, and conditional iteration. In [Exmp. 5](#), we also used the following syntactic sugar:

$$\begin{aligned} \text{foreach}_T x \in E \text{ do } G &\equiv \_iter_T := E ; \\ &\quad \text{while } !\text{isNil}(\_iter_T) \\ &\quad \quad x := \text{headOrElseDefault}(\_iter_T, x) ; G ; \\ &\quad \_iter_T := \text{tailOrElseDefault}(\_iter_T, []) \end{aligned}$$

Here, $\_iter_T$ is a variable that contains a list of data type T .

- Let $\mathbb{K} \rightarrow \mathbb{G}$ denote the set of choreographic *libraries* (partial functions from choreographic procedure names to global programs), ranged over by Ψ .

$$\begin{array}{c}
\frac{\Gamma(x) = T}{\Gamma \vdash x : T} [\vdash\text{-VAR}] \quad \frac{}{\Gamma \vdash p[V_{\text{Nat}}] : \text{Proc}\langle p \rangle} [\vdash\text{-REF}] \quad \frac{}{\Gamma \vdash \text{true} : \text{Bool}} [\vdash\text{-TRUE}] \\
\\
\frac{\Gamma \vdash E : \text{Bool}}{\Gamma \vdash !E : \text{Bool}} [\vdash\text{-NEG}] \quad \dots \quad \frac{\emptyset \vdash \Phi(x) : \Gamma(x) \text{ for each } x \in \text{dom } \Gamma}{\vdash \Phi : \Gamma} [\vdash\text{-STATE}]
\end{array}$$

Fig. 5: Static semantics of data (excerpt)

3.3 Static Semantics

Data Let $\mathbb{X} \rightarrow \mathbb{T}$ denote the set of all *data type environments* (partial functions from variables to data types), ranged over by Γ . Let

$$\Gamma \vdash E : T \quad \text{and} \quad \vdash \Phi : \Gamma$$

denote the *well-typedness* of E by T in Γ , and Φ by Γ . It is the relation induced by the rules in Fig. 5:

- Rules $[\vdash\text{-VAR}]$, $[\vdash\text{-TRUE}]$, and $[\vdash\text{-NEG}]$ are standard [42].
- Rule $[\vdash\text{-REF}]$ states that an indexed point-of-view name $p[V]$ (process reference) is well-typed by $\text{Proc}\langle p \rangle$. Shortly, this judgment will allow us to make assumptions about the presence/absence of variables and choreographic procedures in the configurations of any process $p[V]$ that has point-of-view p .
- Rule $[\vdash\text{-STATE}]$ states that a process state is well-typed when it has a well-typed value for each variable in the data type environment.

Choreographies Let $\mathbb{K} \rightarrow \mathbb{C}$ denote the set of *choreographic type environments* (partial functions from choreographic procedure names to choreographic types), ranged over by Δ . If R is a set of point-of-view names, then let

$$\{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash G : \text{Chor}\langle p \rangle \quad \text{and} \quad \{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash \Psi : \Delta$$

denote the *well-typedness* of G by $\text{Chor}\langle p \rangle$, and Ψ by Δ , in *families* of environments $\{\Gamma_r\}_{r \in R}$ and $\{\Delta_r\}_{r \in R}$. It is the relation induced by the rules in Fig. 6:

- Rule $[\vdash\text{-CALL}]$ states that a choreographic procedure call is well-typed when that procedure exists and is well-typed.
- Rule $[\vdash\text{-NOOP}]$ states that a no-op is well-typed.
- Rule $[\vdash\text{-ASGN}]$ states that an assignment is well-typed from point-of-view p when the variable and the expression are well-typed by the same type in the data type environment of p .
- Rule $[\vdash\text{-COMM}]$ states that a continuation-passing communication is well-typed from point-of-view p (input of the rule, algorithmically) when:
 - the type of the expression from point-of-view p (first premise), and the type of the variable from point-of-view q (third premise), are the same;
 - the receiver has point-of-view q (second premise);
 - the continuation is well-typed from point-of-view q (fourth premise).

Let $\mathbf{\Gamma} = \{\Gamma_r\}_{r \in R}$, let $\mathbf{\Delta} = \{\Delta_r\}_{r \in R}$, and let $p \in R$, s.t. $\mathbf{\Gamma}(p) = \Gamma_p$ and $\mathbf{\Delta}(p) = \Delta_p$.

$$\begin{array}{c}
\frac{\Delta(p)(k) = \mathbf{Chor}\langle p \rangle}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash k : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-CALL}\text{]} \quad \frac{}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{skip} : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-NOP}\text{]} \\
\\
\frac{\mathbf{\Gamma}(p)(x) = T \quad \mathbf{\Gamma}(p) \vdash E : T}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash x := E : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-ASGN}\text{]} \\
\\
\frac{\mathbf{\Gamma}(p) \vdash E_1 : T \quad \mathbf{\Gamma}(p) \vdash E_2 : \mathbf{Proc}\langle q \rangle \quad \mathbf{\Gamma}(q)(y) = T \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G : \mathbf{Chor}\langle q \rangle}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash E_1 \rightarrow E_2.y \triangleright G : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-COMM}\text{]} \\
\\
\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 : \mathbf{Chor}\langle p \rangle \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_2 : \mathbf{Chor}\langle p \rangle}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 ; G_2 : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-SEQ}\text{]} \\
\\
\frac{\Gamma_p \vdash E : \mathbf{Bool} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_1 : \mathbf{Chor}\langle p \rangle \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G_2 : \mathbf{Chor}\langle p \rangle}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{if } E \mathbf{ } G_1 \mathbf{ } G_2 : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-IF}\text{]} \\
\\
\frac{\Gamma_p \vdash E : \mathbf{Bool} \quad \mathbf{\Gamma}, \mathbf{\Delta} \vdash G : \mathbf{Chor}\langle p \rangle}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \mathbf{while } E \mathbf{ } G : \mathbf{Chor}\langle p \rangle} \text{[}\vdash\text{-WHILE}\text{]} \\
\\
\frac{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \Psi(k) : \Delta(k) \text{ for each } k \in \text{dom } \Delta}{\mathbf{\Gamma}, \mathbf{\Delta} \vdash \Psi : \Delta} \text{[}\vdash\text{-LIB}\text{]}
\end{array}$$

Fig. 6: Static semantics of choreographies

$$\begin{array}{ll}
\Phi[x] = \Phi(x) & \Phi[!E] = \begin{cases} \mathbf{true} & \text{if } \Phi[E] = \mathbf{false} \\ \mathbf{false} & \text{otherwise} \end{cases} \quad \dots \\
\Phi[V] = V &
\end{array}$$

Fig. 7: Dynamic semantics of data (excerpt)

This rule is the reason why *families* of data/choreographic type environments are needed in the judgments instead of individual environments: as we “cascade” along continuation-passing communications, at each next level, different environments are needed than at the previous level.

- Rules [\[}\vdash\text{-SEQ}\text{\]}](#), [\[}\vdash\text{-IF}\text{\]}](#), and [\[}\vdash\text{-WHILE}\text{\]}](#) are straightforward.
- Rule [\[}\vdash\text{-LIB}\text{\]}](#) states that a choreographic library is well-typed when it has a well-typed global program for each name in the library.

We note that checking well-typedness is decidable and, in fact, linear in the size of the global program (because the typing rules are syntax-directed).

3.4 Dynamic Semantics

Data Let $\Phi[E]$ denote the *evaluation* of E in Φ . It is the smallest function induced by the equations in [Fig. 7](#). We note that the function is only partially defined. For instance, $\Phi[x]$ and $\Phi[!123]$ are undefined. Undefinedness is potentially problematic: it causes processes to get stuck, which in turn causes com-

$$\begin{array}{c}
\frac{}{(\Phi, \Psi, \mathbf{skip}) \downarrow} [\downarrow\text{-NoOp}] \quad \frac{(\Phi, \Psi, G_1) \downarrow \quad (\Phi, \Psi, G_2) \downarrow}{(\Phi, \Psi, G_1 ; G_2) \downarrow} [\downarrow\text{-SEQ}] \\
\\
\frac{\Psi(k) = G}{(\Phi, \Psi, k) \xrightarrow{\tau} (\Phi, \Psi, G)} [\rightarrow\text{-CALL}] \quad \frac{\Phi\{x \mapsto \Phi[E]\} = \Phi'}{(\Phi, \Psi, x := E) \xrightarrow{\tau} (\Phi', \Psi, \mathbf{skip})} [\rightarrow\text{-ASGN}] \\
\\
\frac{\Phi[E_1] = V \quad \Phi[E_2] = q[j]}{(\Phi, \Psi, E_1 \rightarrow E_2.y \triangleright G) \xrightarrow{q[j]!(y := V ; G)} (\Phi, \Psi, \mathbf{skip})} [\rightarrow\text{-SEND}] \\
\\
\frac{}{(\Phi, \Psi, G_1) \xrightarrow{p[i]?G_2} (\Phi, \Psi, G_1 ; G_2)} [\rightarrow\text{-RECV}] \\
\\
\frac{(\Phi, \Psi, G_1) \xrightarrow{A} (\Phi', \Psi, G'_1) \quad A \neq p[i]?G \text{ for each } p[i], G}{(\Phi, \Psi, G_1 ; G_2) \xrightarrow{A} (\Phi', \Psi, G'_1 ; G_2)} [\rightarrow\text{-SEQ1}] \\
\\
\frac{(\Phi, \Psi, G_1) \downarrow \quad (\Phi, \Psi, G_2) \xrightarrow{A} (\Phi', \Psi, G'_2) \quad A \neq p[i]?G \text{ for each } p[i], G}{(\Phi, \Psi, G_1 ; G_2) \xrightarrow{A} (\Phi', \Psi, G'_2)} [\rightarrow\text{-SEQ2}] \\
\\
\frac{\Phi[E] = \mathbf{true}}{(\Phi, \Psi, \mathbf{if } E \text{ } G_1 \text{ } G_2) \xrightarrow{\tau} (\Phi, \Psi, G_1)} [\rightarrow\text{-IF1}] \quad \frac{\Phi[E] = \mathbf{false}}{(\Phi, \Psi, \mathbf{if } E \text{ } G_1 \text{ } G_2) \xrightarrow{\tau} (\Phi, \Psi, G_2)} [\rightarrow\text{-IF2}] \\
\\
\frac{\Phi[E] = \mathbf{true}}{(\Phi, \Psi, \mathbf{while } E \text{ } G) \xrightarrow{\tau} (\Phi, \Psi, G ; \mathbf{while } E \text{ } G)} [\rightarrow\text{-WHILE1}] \quad \frac{\Phi[E] = \mathbf{false}}{(\Phi, \Psi, \mathbf{while } E \text{ } G) \xrightarrow{\tau} (\Phi, \Psi, \mathbf{skip})} [\rightarrow\text{-WHILE2}]
\end{array}$$

Fig. 8: Dynamic semantics of choreographies (part 1)

munication deadlocks among processes. However, using the static semantics of the previous section, undefinedness at run-time is detected at compile-time.

Choreographies Let $\mathbb{A}$ denote the set of *actions* (labels), ranged over by A :

$$A ::= q[j]!G \mid p[i]?G \mid \tau$$

Actions $q[j]!G$ and $p[i]?G$ define a *send* and a *receive* of global program G from process $p[i]$ to process $q[j]$. In this paper, G is of the form $y := V ; G'$, where V and G' are a message and a continuation. Action τ defines an internal action.

Let $(\Phi, \Psi, G) \downarrow$ and $(\Phi, \Psi, G) \xrightarrow{A} (\Phi', \Psi', G')$ denote the *termination* of configuration (Φ, Ψ, G) and the *reduction* of source configuration (Φ, Ψ, G) to target configuration (Φ', Ψ', G') through action A . They are the relations induced by the rules in Fig. 8:

- Rule $[\downarrow\text{-NoOp}]$ states that a no-op can terminate.
- Rule $[\downarrow\text{-SEQ}]$ states that sequencing can terminate when the operands can.
- Rule $[\rightarrow\text{-CALL}]$ states that a call can reduce to the body of the corresponding choreographic procedure through an internal action.

- Rule $[\rightarrow\text{-ASGN}]$ states that an assignment can reduce to a no-op through an internal action, while the state in the configuration is updated accordingly.
- Rule $[\rightarrow\text{-SEND}]$ states that a continuation-passing communication can reduce to a no-op through a send. The outgoing global program contains the message and the continuation. In this way, continuation-passing is a form of *process-driven* dynamic projection: the sender at run-time tells the receiver how to behave (from its first-person point-of-view), instead of a choreographic programming tool at compile-time (from its third-person point-of-view).
- Rule $[\rightarrow\text{-RECV}]$ states that *any* global program can reduce through a receive. The first operand of the resulting sequence is the original global program; the second operand is the incoming global program, which contains a message and a continuation. In this way, the incoming global program is explicitly scheduled to be run, but only when all existing work is done. Thus, scheduling is non-preemptive, reminiscent of event-driven programming.

We note that there is a difference between: **(1)** the “physical receive” that takes the message/continuation off-network (happens when rule $[\rightarrow\text{-RECV}]$ is applied, and G_2 is appended to G_1); **(2)** the “logical receive” that actually processes the message/continuation (happens when the reduction of G_1 is finished, and the reduction of G_2 is started). Because of this decoupling, no explicit receive operations need to be defined in global programs: they happen implicitly, at any time, induced by the dynamic semantics (in practice, the run-time system takes care of this). This is reminiscent of: **(1)** buffering a message in a local event queue; **(2)** taking that message out of the event queue in a later iteration of the event loop to process it.

- Rules $[\rightarrow\text{-SEQ1}]$ and $[\rightarrow\text{-SEQ2}]$ are standard in the dynamic semantics of imperative languages [1] and process algebra [22], except that receives are excluded. Thus, receives happen only at the top-level and cannot preempt.
- Rules $[\rightarrow\text{-IF1}]$, $[\rightarrow\text{-IF2}]$, $[\rightarrow\text{-WHILE1}]$, and $[\rightarrow\text{-WHILE2}]$ are standard in the dynamic semantics of imperative languages [1] and process algebra [22].

We note that choreographic procedures enhance expressiveness relative to conditional iteration: not every recursive call can be simulated with **while**. The difference is that each iteration of **while** must start at the same process (the active process in the loop’s enclosing scope), while each recursive call may start at a different process (the active process in the call site’s enclosing scope).

If Π is a set of indexed point-of-view names, ranged over by π , then let

$$\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \downarrow \quad \text{and} \quad \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \rightarrow \{(\Phi'_\pi, \Psi'_\pi, G'_\pi)\}_{\pi \in \Pi}$$

denote the *termination* and *reduction of families* of configurations. That is, each configuration models a process, while the family models the distributed system. Formally, $\downarrow$ and $\rightarrow$ are the relations induced by the rules in Fig. 9:

- Rule $[\downarrow\text{-SYS}]$ states that a family can terminate when each of its configurations can terminate. This rule is standard in the dynamic semantics of parallel composition in process algebra [22].

$$\begin{array}{c}
\frac{(\Phi_\pi, \Psi_\pi, G_\pi) \downarrow \text{ for each } \pi \in \Pi}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \downarrow} \quad [\downarrow\text{-Sys}] \\
\\
\frac{(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{\tau} C'_{p[i]} \quad p[i] \in \Pi}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \rightarrow \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi \setminus \{p[i]\}} \cup \{C'_\pi\}_{\pi \in \{p[i]\}}} \quad [\rightarrow\text{-Sys1}] \\
\\
\frac{
\begin{array}{c}
(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{q[j]!G} C'_{p[i]} \quad p[i] \in \Pi \\
(\Phi_{q[j]}, \Psi_{q[j]}, G_{q[j]}) \xrightarrow{p[i]?G} C'_{q[j]} \quad q[j] \in \Pi \setminus \{p[i]\}
\end{array}
}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \rightarrow \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi \setminus \{p[i], q[j]\}} \cup \{C'_\pi\}_{\pi \in \{p[i], q[j]\}}} \quad [\rightarrow\text{-Sys2}] \\
\\
\frac{
\begin{array}{c}
(\Phi_{p[i]}, \Psi_{p[i]}, G_{p[i]}) \xrightarrow{p[i]!G} C'_{p[i]} \xrightarrow{p[i]?G} C''_{p[i]} \quad p[i] \in \Pi \\
\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \rightarrow \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi \setminus \{p[i]\}} \cup \{C''_\pi\}_{\pi \in \{p[i]\}}
\end{array}
}{\{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi} \rightarrow \{(\Phi_\pi, \Psi_\pi, G_\pi)\}_{\pi \in \Pi \setminus \{p[i]\}} \cup \{C''_\pi\}_{\pi \in \{p[i]\}}} \quad [\rightarrow\text{-Sys3}]
\end{array}$$

Fig. 9: Dynamic semantics of choreographies (part 2)

- Rule $[\rightarrow\text{-Sys1}]$ states that a family can reduce when one of its configurations can reduce through an internal action. This rule models independent evolution of processes in the absence of communication dependencies.
- Rule $[\rightarrow\text{-Sys2}]$ states that a family can reduce when one of its configurations can reduce through a send, while another configuration can reduce through a corresponding receive. This rule models communication between processes. We note that the send and the *physical* receive are synchronous (as usual in choreographic programming), but the *logical* receive is asynchronous.
- Rule $[\rightarrow\text{-Sys3}]$ states that a family can reduce when one of its configurations can reduce through a self-communication (first a send, next the receive, by one process). This allows a process to schedule work for itself for a later time. We note that the (self-)send and the physical (self-)receive are, like in rule $[\rightarrow\text{-Sys2}]$, effectively synchronous (i.e., the two reductions in the premise result in a single reduction in the conclusion), but they need to be ordered.

3.5 Main Result: Communication Deadlock Freedom by Design

Soundness The following theorem states our main result. To formulate it, let $\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi}$ denote an initial family of configurations, where:

- Π denotes a set of indexed point-of-view names (process references);
- $\Phi_{p[i]}$ denotes an initial state for each $p[i] \in \Pi$;
- Ψ_p and G_p denote a choreographic library and an initial global program for each point-of-view $p \in \{p \mid p[i] \in \Pi\}$ (i.e., processes of the same point-of-view, have the same choreographic library and initial global program).

Now, the initial family of configurations either reduces finitely many times and terminates, or it reduces infinitely many times (i.e., it never gets stuck, so we enjoy communication deadlock freedom), when the following conditions are met:

1. Each choreographic library Ψ_p is well-typed. This is a static check.
2. Each initial global program G_p is well-typed. This is a static check.

3. Each initial state $\Phi_{p[i]}$ is well-typed. This is a lightweight dynamic check, part of initialisation (input validation).
4. The family of configurations is *closed*: if a process is referenced in a configuration, then it must have a configuration in the family. This is a lightweight dynamic check, part of initialisation (network validation). In the theorem below, let $\text{refs}(\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi})$ denote the set of process references that occur in any configuration of the family.

Intuitively, the static checks ensure that, if the initial configurations are proper, then a process never passes a continuation to another process that cannot run to completion: well-typedness guarantees that *expected* variables of *expected* types, as well as *expected* choreographic procedures, *actually* exist. Complementarily, the lightweight dynamic checks ensure that the initial configurations are proper.

Theorem 1. *Let $R = \{p \mid p[i] \in \Pi\}$. Assume:*

1. **Statically:** $\{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash \Psi_p : \Delta_p$ for each $p \in R$
2. **Statically:** $\{\Gamma_r\}_{r \in R}, \{\Delta_r\}_{r \in R} \vdash G_p : \text{Chor}\langle p \rangle$ for each $p \in R$
3. **Dynamically:** $\vdash \Phi_{p[i]} : \Gamma_p$ for each $p[i] \in \Pi$
4. **Dynamically:** $\text{refs}(\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi}) \subseteq \Pi$

If $\{(\Phi_{p[i]}, \Psi_p, G_p)\}_{p[i] \in \Pi} \rightarrow^* C'$, then either $C' \downarrow$, or $C' \rightarrow C''$, for some C'' .

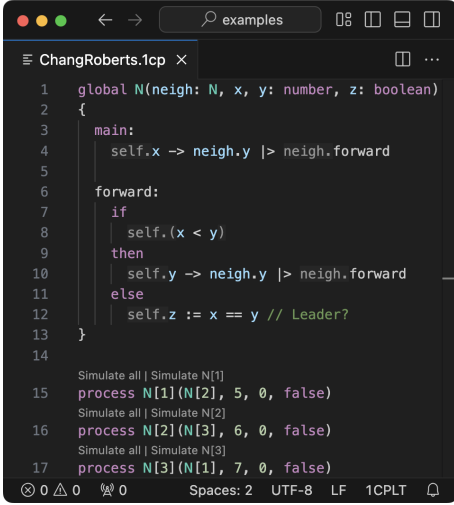
We formalised all the definitions of this section in Isabelle/HOL and mechanised the proof of this theorem and all auxiliary lemmas. The following outline summarises the main steps and structure. See the Isabelle/HOL code for details. Extending this result to livelock is left for future work.

Proof (outline). The proof of [Thm. 1](#) is split into two parts: *progress* and *preservation*. Progress states that if the four assumption hold for a family, then that family can either terminate or reduce. Preservation states that if the four assumptions hold for a source family, and if that source family reduces to a target family, then the four assumptions also hold for the target family. Thus, if the four assumptions hold for the initial family, then by applying progress and preservation inductively, we obtain the main result in [Thm. 1](#). The proofs of progress and preservation use several forms of induction (on the structure of the terms; on the derivation of the judgments), nested inside each other through lemmas. $\square$

Expressiveness Expressiveness of choreographic programming calculi can be measured along (at least) two relevant dimensions:

- Computability: Which classes of functional behaviour can be expressed?
- Message complexity: How many messages are needed and in which patterns?

Regarding computability, it has been shown that a simple core calculus of choreographies is Turing-complete (e.g., [16]). That calculus can be trivially embedded in ours. Regarding message complexity, none of the existing choreographic programming languages we know of supports all realisable *regular* message patterns



$$\begin{aligned}
\Gamma_{\mathbf{N}} &= \{\text{neigh} \mapsto \text{Proc}[\mathbf{N}], x \mapsto \text{Nat}, \dots\} \\
G_{\mathbf{N}} &= x \rightarrow \text{neigh.y} \triangleright \text{forward} \\
\Psi_{\mathbf{N}} &= \left\{ \begin{array}{l} \text{forward} \mapsto \text{if } x < y \\ \qquad \qquad \qquad y \rightarrow \text{neigh.y} \triangleright \\ \qquad \qquad \qquad \text{forward} \\ \qquad \qquad \qquad z := x == y \end{array} \right\} \\
&= \text{--- static part } \Delta \text{ --- dynamic part } \nabla \text{ ---} \\
\Phi_{\mathbf{N}[1]} &= \{\text{neigh} \mapsto \mathbf{N}[2], x \mapsto 5, \dots\} \\
\Phi_{\mathbf{N}[2]} &= \{\text{neigh} \mapsto \mathbf{N}[3], x \mapsto 6, \dots\} \\
\Phi_{\mathbf{N}[3]} &= \{\text{neigh} \mapsto \mathbf{N}[1], x \mapsto 7, \dots\}
\end{aligned}$$

Fig. 10: Correspondence between 1CPLT and the calculus

(i.e., message patterns that can be specified with a regular expression over synchronous communications; e.g., M-out-of-N for any concrete m and n). In contrast, in addition to supporting parametricity, we conjecture that our approach also supports all realisable regular message patterns.

4 Implementation: 1CPLT

4.1 From Theory to Practice

The calculus presented in the previous section is expressive, but it(s notation) is unwieldy to use directly to implement distributed systems. With the development of 1CPLT, demonstrated in Sect. 2.2, we set out to explore how to possibly transfer the calculus from theory to practice, in a way that intuitively and concisely hides its complexities; see also the artifact [35].

Fig. 10 demonstrates the correspondence between the concrete syntax of 1CPLT and the ingredients of the calculus. Inspired by classes in object-oriented programming, the idea is to combine, for each point-of-view, the declarations of variables and the definitions of choreographic procedures in the same “unit” (global block), using a special choreographic procedure (main) to define the initial global program. This is all the information needed for static type checking. For simulation purposes, the creation of configurations is expressed separately (process statements) by defining the initial values of variables for each process.

4.2 Technical Details

General 1CPLT is implemented in *Rascal* [38], a meta-programming environment geared towards implementing programming languages. Among other fea-

tures, the *Rascal language* has core support to write context-free grammars (for defining concrete syntax), algebraic data types (for defining abstract syntax), and advanced pattern matching on grammar rules and ADT constructors. Together with standard programming abstractions, these features aim to simplify the implementation of parsers, type checkers, interpreters, and code generators.

Parsing The parser consumes a `.1cp` file as input and produces as output:

- for each point-of-view: abstract syntax trees for a data type environment, a choreographic type environment, and a choreographic library (`global` block);
- for each process: abstract syntax trees for an initial state (`process stmtnt`).

The parser proceeds in two steps:

1. First, it transforms the input file into a concrete syntax tree using the *1CPLT grammar*. For instance, the following *production rules* in the 1CPLT grammar define the concrete syntax of assignments and communications (simplified to save space; see the artifact for the full version):

```

1 syntax ChorExpression
2   = ...
3   | Identifier      ":" DataExpression
4   | DataExpression "->" DataExpression "." Identifier ">" ChorExpression
5   | ... ;

```

2. Next, the parser transforms the concrete syntax tree into an abstract version using algebraic data types. For instance, the following *constructors* define the abstract syntax of assignments and continuation-passing communications (simplified to save space; see the artifact for the full version):

```

1 data CHOR_EXPRESSION(loc src = |unknown:///|)
2   = ...
3   | asgn(str x, DATA_EXPRESSION e)
4   | comm(DATA_EXPRESSION e, DATA_EXPRESSION qj, str y, CHOR_EXPRESSION c)
5   | ... ;

```

We note that each constructor has an optional argument `src` (declared on line 1), which associates each value of a constructor with a particular location in the input file (`loc` is a core data type in Rascal for locations). This is essential for the type checker to generate properly located error messages.

Type checking The type checker consumes abstract syntax trees as input and produces a list of error messages as output; the list is empty if, and only if, the input is well-typed. Essentially, the type checker implements the typing rules in the static semantics of the calculus. For instance, the following code type-checks `asgn` constructors (simplified to save space; see the artifact for the full version):

```

1 data CHOR_CONTEXT = context(
2   map[str /* pov */, map[str /* data variables */, DATA_TYPE]] gammas,
3   map[str /* pov */, map[str /* chor variables */, CHOR_TYPE]] deltas) ;
4
5 list[Message] check(chor(p), CHOR_CONTEXT c, tree: asgn(x, e))
6   = [error("Unexpected data variable", tree.src) | x notin c.gammas[p]]
7   + [*check(c.gammas[p][x], context(c.gammas[p]), e) | x in c.gammas[p]] ;

```

For reference and comparison, typing rule $\llbracket \vdash \text{ASGN} \rrbracket$ (Fig. 6) is reproduced below.

Lines 1–3 define a general ADT for choreographic type environments. Line 5 declares the arguments to type-check `asgn` constructors, each of which corresponds with a variable in the conclusion of the typing rule: p with p , c with Γ and Δ , x with x , and e with e ; `tree` is just a shorthand for `asgn(x, e)`.

Line 6 checks if x exists in the data type environment of p ; if not, an error message is generated. The second argument of constructor `error` is the location in the input file where the error message should be displayed. Line 7 recursively calls function `check` to collect all error messages that result from type-checking data expression e (E) against data type $c.\text{gammass}[p][x]$ ($\Gamma(p)(x)$) in data environment $c.\text{gammass}[p]$ ($\Gamma(p)$). This is expressed using Rascal’s notation for list comprehension: $[x \mid P(x), Q(x), \dots]$ constructs a list such that each x satisfies the conjunction of $P(x)$, $Q(x)$, ... Operator $*$ splices a list into another list.

The functions to type-check the other constructors are defined similarly.

Interpretation The interpreter consumes abstract syntax trees as input (initial states and choreographic libraries) and produces a simulation as output. Essentially, the interpreter implements the reduction rules of the dynamic semantics of the calculus. For instance, the following code reduces `asgn` constructors (simplified to save space; see the artifact for the full version):

```
1 tuple[CHOR_STATE, CHOR_EXPRESSION] reduce(<state(phi, psi, {}), asgn(x, e)>)
2 = <state(phi + (x: eval(phi, e)), psi, {}), skip()> ;
```

Line 1 declares that function `reduce` consumes a source configuration as input (i.e., the arguments are directly destructured using pattern matching). Line 2 defines that function `reduce` produces a target configuration as output by updating the value of x in phi , in accordance with rule $[\rightarrow\text{-ASGN}]$ (Fig. 8).

The functions to reduce the other constructors are defined similarly.

Before each simulation, it is also checked whether or not each initial process state is well-typed, as well as closedness (points 3–4 of Thm. 1).

Code generation (outlook) We aim to extend 1CPLT with a code generator to be able to deploy 1CPLT implementations on real networks.

The most interesting aspect of such a code generator is that it should “compile away” continuation-passing: while continuation-passing is instrumental in our approach *as a programming abstraction*, physically passing around code-to-be-executed tends to be problematic in terms of security, privacy, and performance. Instead, we envisage a static transformation in which: **(a)** for each continuation, a choreographic procedure is introduced with a random label; **(b)** instead of passing the continuation, only the random label is passed.

5 Conclusion

5.1 Related Work

Process-parametric MPST Close to the work in this paper, in terms of starting point and aim, are several extensions of *multiparty session typing* (MPST)

[30] to support process-parametricity. First, Deniérou et al. [51] designed a dependently typed calculus to support process-parametricity, but the resulting typing relation is undecidable and was never implemented. Next, taking the opposite approach, Ng and Yoshida [41] implemented an extension of the *Scribble* tools for MPST [31, 32], called *Pabble*, to support process-parametricity, but it was never formalised and does not provide type-soundness guarantees. Last, Castro et al. [8] presented a calculus (all static analysis decidable) and a tool (formalised by the calculus) to support process-parametricity, but the expressiveness is limited (many distributed systems remain unsupported, including those in Sect. 2.2), while the static analysis algorithm is of exponential complexity. Thus, there is tension between decidability, soundness, and expressiveness.⁴ However, by developing process-driven dynamic projection through continuation-passing communications (which diverges considerably from classical static MPST), our approach in this paper manages to combine these properties.

Dynamic projection Shen et al. [50] developed a choreographic programming library in Haskell, called *HasChor*, that uses dynamic projection to be able to build their library on top of the *freer monad* [37]. In this way, dynamic projection is primarily an implementation mechanism in *HasChor*. *HasChor* does not support process-parametricity, and it has not been formalised yet.

In MPST, dynamic projection has been studied by Hamers et al. [27, 34] in the form of a run-time verification framework based on global types, called *Discourje*. *Discourje* also supports process-parametricity, but it imposes run-time overhead, as all analysis happens dynamically. In contrast, the analysis in this paper happens almost completely statically.

Behavioural-typed actors/mailboxes The event-driven nature of our approach to choreographic programming is reminiscent of programming methods based on event loops/queues and asynchronous execution of event handlers. Closest in this area is the work on behavioural typing of actors and mailboxes by Fowler et al. [23, 24], De'Liguoro and Padovani [21], and Scalas et al. [48].

More fundamentally, we speculate that a deeper connection may be uncovered between actors/mailboxes and the approach in this paper. Our initial observation is that a mailbox-based runtime for 1CPLT (in combination with code generation; end of Sect. 4.2) would align perfectly with the dynamic semantics of our calculus—much more so than a (point-to-point) channel-based runtime would. This leads us to hypothesise that mailbox-based actors are to first-person choreographies what (point-to-point) channel-based processes are to third-person choreographies. That is, the following table would be completed:

⁴ While we are not aware of any formal results about inherent limitations of static projection, we strongly suspect such limitations exist (because, fundamentally, fewer information is available at compile-time than at run-time).

global model of concurrency (at development-time)	(third-person) choreographies	??? $\Leftarrow$ 1CP
local model of concurrency (at deployment-time)	processes + channels	actors + mailboxes
behavioural typing discipline	session types	mailbox types

Proof mechanisation In recent years, mechanisation of choreographic programming theory has received considerable attention, including the work of Cruz-Felipe et al. [15, 17–20], Hirsch and Garg [29], and Pahjola et al. [44]. Similar initiatives for behavioural typing include the work of Castro et al. [9, 10], Hinrichson et al. [28], and Jacobs et al. [33]. Thus, the mechanisation of our proofs in Isabelle/HOL follows this trend in our community.

5.2 Future Work

On the theoretical side, we identify two directions for future work. First, we are keen to investigate the integration of dependent typing into our approach, to support arbitrary expressions as indices of point-of-view names (process references) instead of only values. Second, it will be interesting to develop dedicated testing and verification techniques for first-person choreographies, following recent work for third-person choreographies (e.g., [12, 13, 36]).

Another interesting question is whether a first-person approach could also be applied in MPST. Most MPST theories have a tight correspondence between the syntax of global/local types and the syntax of processes. If this correspondence is kept tight (i.e., a first-person approach is used both in global/local types and in processes), then we expect it to be possible—but still non-trivial—to develop typing rules accordingly and prove soundness. These typing rules would not rely on projection, but verify processes directly against global types. An alternative (presumably more challenging, but arguably more usable) would be to develop an MPST theory in which the correspondence is loosened: combining a first-person approach in global types with a more conventional style for processes.

On the practical side, in combination with an extension of 1CPLT with a code generator (Sect. 4.2), we are eager to conduct a case study on *Paxos*. This is a popular (family of) distributed algorithm(s) for *consensus*. Paxos has a number of properties (particularly related to non-determinism) that are fundamentally hard to support with third-person choreographies. We believe first-person choreographies can address these issues.

Data Availability Statement

The artifact is available on Zenodo [35]. It contains: (1) the calculus in Isabelle/HOL; (2) the language and tooling in VS Code.

References

1. Apt, K.R., de Boer, F.S., Olderog, E.: *Verification of Sequential and Concurrent Programs*. Texts in Computer Science, Springer (2009)
2. Carbone, M., Cruz-Filipe, L., Montesi, F., Murawska, A.: Multiparty classical choreographies. In: LOPSTR. Lecture Notes in Computer Science, vol. 11408, pp. 59–76. Springer (2018)
3. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: ESOP. Lecture Notes in Computer Science, vol. 4421, pp. 2–17. Springer (2007)
4. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* **34**(2), 8:1–8:78 (2012)
5. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274. ACM (2013)
6. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. In: CONCUR. Lecture Notes in Computer Science, vol. 8704, pp. 47–62. Springer (2014)
7. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. *Distributed Comput.* **31**(1), 51–67 (2018)
8. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *PACMPL* **3**(POPL), 29:1–29:30 (2019)
9. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zoooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: PLDI, pp. 237–251. ACM (2021)
10. Castro-Perez, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: TACAS (2). Lecture Notes in Computer Science, vol. 12079, pp. 278–285. Springer (2020)
11. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extremafinding in circular configurations of processes. *Commun. ACM* **22**(5), 281–283 (1979)
12. Coto, A., Guanciale, R., Tuosto, E.: An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.* **123**, 100712 (2021)
13. Cruz-Filipe, L., Graversen, E., Montesi, F., Peressotti, M.: Reasoning about choreographic programs. In: COORDINATION. Lecture Notes in Computer Science, vol. 13908, pp. 144–162. Springer (2023)
14. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: FoSSaCS. Lecture Notes in Computer Science, vol. 10203, pp. 424–440 (2017)
15. Cruz-Filipe, L., Lugovic, L., Montesi, F.: Certified compilation of choreographies with hacc. In: FORTE. Lecture Notes in Computer Science, vol. 13910, pp. 29–36. Springer (2023)
16. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. *Theor. Comput. Sci.* **802**, 38–66 (2020)
17. Cruz-Filipe, L., Montesi, F.: Now it compiles! certified automatic repair of un-compilable protocols. In: ITP. LIPIcs, vol. 268, pp. 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
18. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: ICTAC. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer (2021)

19. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: ITP. LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
20. Cruz-Filipe, L., Montesi, F., Peressotti, M.: A formal theory of choreographic programming. *J. Autom. Reason.* **67**(2), 21 (2023)
21. de'Liguoro, U., Padovani, L.: Mailbox types for unordered interactions. In: ECOOP. LIPIcs, vol. 109, pp. 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
22. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series, Springer (2000)
23. Fowler, S., Attard, D.P., Sowul, F., Gay, S.J., Trinder, P.: Special delivery: Programming with mailbox types. *Proc. ACM Program. Lang.* **7**(ICFP), 78–107 (2023)
24. Fowler, S., Lindley, S., Wadler, P.: Mixing metaphors: Actors as channels and channels as actors. In: ECOOP. LIPIcs, vol. 74, pp. 11:1–11:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
25. Giallorenzo, S., Montesi, F., Peressotti, M.: Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.* **46**(1), 1:1–1:59 (2024)
26. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: The choreographic and multitier cases (pearl). In: ECOOP. LIPIcs, vol. 194, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
27. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in clojure. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 266–284. Springer (2020)
28. Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* **4**(POPL), 6:1–6:30 (2020)
29. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* **6**(POPL), 1–27 (2022)
30. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM (2008)
31. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer (2016)
32. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer (2017)
33. Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.* **6**(ICFP), 466–495 (2022)
34. Jongmans, S.: Discourje: Run-time verification of communication protocols in clojure - live at last. In: FM (2). Lecture Notes in Computer Science, vol. 14934, pp. 158–166. Springer (2024)
35. Jongmans, S.: First-person choreographic programming with continuation-passing communications (artifact) (2025), <https://doi.org/10.5281/zenodo.14624355>
36. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies - computing preconditions in choreographic programming. In: ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 520–547. Springer (2022)
37. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Haskell, pp. 94–105. ACM (2015)
38. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with rascal. In: GTTSE. Lecture Notes in Computer Science, vol. 6491, pp. 222–289. Springer (2009)

39. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023)
40. Montesi, F., Yoshida, N.: Compositional choreographies. In: CONCUR. Lecture Notes in Computer Science, vol. 8052, pp. 425–439. Springer (2013)
41. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Service Oriented Computing and Applications **9**(3-4), 269–284 (2015)
42. Pierce, B.C.: Types and programming languages. MIT Press (2002)
43. Plyukhin, D., Peressotti, M., Montesi, F.: Ozone: Fully out-of-order choreographies. In: ECOOP. LIPIcs, vol. 313, pp. 31:1–31:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024)
44. Pohjola, J.Å., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: A verified, end-to-end compiler for a choreographic language. In: ITP. LIPIcs, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
45. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies - safe runtime updates of distributed applications. In: COORDINATION. Lecture Notes in Computer Science, vol. 9037, pp. 67–82. Springer (2015)
46. Preda, M.D., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: Theory and implementation. Log. Methods Comput. Sci. **13**(2) (2017)
47. Preda, M.D., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: A choreographic framework for safe adaptive distributed applications. In: SLE. Lecture Notes in Computer Science, vol. 8706, pp. 161–170. Springer (2014)
48. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: PLDI. pp. 502–516. ACM (2019)
49. Segall, A.: Distributed network protocols. IEEE Trans. Inf. Theory **29**(1), 23–34 (1983)
50. Shen, G., Kashiwa, S., Kuper, L.: Haschor: Functional choreographic programming for all (functional pearl). Proc. ACM Program. Lang. **7**(ICFP), 541–565 (2023)
51. Yoshida, N., Deniérou, P., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: FoSSaCS. Lecture Notes in Computer Science, vol. 6014, pp. 128–145. Springer (2010)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Formal Verification of WTO-based Dataflow Solvers

Roméo La Spina , Delphine Demange , and Sandrine Blazy ^(✉) 

Univ Rennes, Inria, CNRS, IRISA, Rennes, France
`{romeo.la-spina,delpnine.demange,sandrine.blazy}@irisa.fr`

Abstract. In this paper, we consider specific dataflow solvers, inspired by the work of Bourdoncle [5], in which an iteration order is pre-computed, based on the structure of the control-flow graph of programs. Our work aims at a clearer formulation and a better understanding of the folklore algorithms proposed three decades ago.

We present a formalization of the dataflow solvers. Central to the proof of correctness is the general notion of a Weak Topological Ordering (WTO). Our correctness proofs are valid for any such ordering. The first solver implements an iterative strategy over the ordering, the second solver implements a recursive strategy.

Our formalization is done within the Coq proof assistant and our solvers are extractable to OCaml code. Our formalization is fully compatible with the interface of dataflow solvers of the verified, optimizing C CompCert compiler. We conduct practical experiments on the wide range of forward and backward analyses of CompCert, demonstrating the practicality of our solvers in terms of efficiency and precision.

Keywords: static analysis · verified compilation · compiler optimization

1 Introduction

Dataflow analysis for program analysis dates back to the end of the 70's with Allen et al.'s work, at the time where the first program optimizations for PL/I compilers were developed. Dataflow analysis is now a well-established compiler basic brick, on which compilers rely to analyze, in a sound and static way, facts or information about the values computed by programs at run-time.

In the dataflow analysis framework, program properties are expressed as a solution of a system of mutually recursive (in)equations, specifying how the program abstract state is modified by each program construct. Given such an equation system, a solution is then effectively computed using a dataflow solver, which computes a fixpoint of the equations. Depending on the mathematical structure underlying the information of interest, the fixpoint is not always unique.

As of today in the compiler construction literature and state-of-the-art, the algorithm proposed by Kildall [13] and variants thereof constitute the de facto standard to implement dataflow solvers. It is part of the general family of iterative dataflow solvers, where, starting from an initial empty information, dataflow

equations are applied iteratively, until a sound, over-approximating fixpoint solution is reached for all equations. The algorithm uses a worklist to keep track of the dataflow equations that remain to be solved, and a fixpoint is considered to be reached when that worklist is empty. This algorithm is simple to implement and easy to understand. Its theoretical foundations have been extensively studied, up to the point that it has been formally verified in the Coq proof assistant, and implemented in the CompCert verified C compiler [19]. In practice, however, one must be careful on the particular choice of the data-structure used to implement the worklist: even though the correctness of the algorithm does not depend on it, the convergence speed crucially depends on the order in which the equations are picked from the worklist.

Intrinsically, Kildall’s algorithm treats dependencies between dataflow equations dynamically: the worklist evolves along the iterations on the equations, and applying an equation can trigger the (future) re-application of some other equations. Choosing the right implementation of the worklist, and pre-computing a partial information on the dependencies between the equations thus allows to circumvent this dynamic treatment of dependencies, and is key to ensure satisfying convergence speeds. Typically, CompCert relies on a reverse postorder numbering of the nodes of the control-flow graph to reflect potential dependencies between the dataflow equations of program instructions. The worklist is implemented with a heap, from which one can pick the next node, and thus the corresponding equations to apply, in an appropriate way, based on its numbering.

Many other iterative dataflow solving algorithms and techniques exist. In particular, a number of them are designed to process the dataflow equations in an order that is statically determined: the solver iteratively applies the equations following this static order. In this paper, we study one such particular technique, proposed by Bourdoncle [5]. This technique originates from the abstract interpretation community, where abstract domains and transfer functions are usually considered to be computationally expensive, and where sophisticated widening techniques are employed to guarantee the convergence of the algorithm. The proposed algorithmic methods implement efficient chaotic iteration strategies, both in terms of precision and performance.

At the heart of the solving methods proposed by Bourdoncle is the notion of a *weak topological ordering* (WTO) over the nodes of the graph, pre-computed based on the structure of the program. For a given static ordering, Bourdoncle proposes two flavors of solver algorithms, each with a different strategy to iteratively apply the equations: the iterative strategy and the recursive strategy. Since their inception, Bourdoncle’s solving methods have been implemented in the vast majority of static analyzers based on abstract interpretation.

From a formal proof perspective, however, those solvers have received little attention. The solver algorithms are not exposed and just described with prose, leaving the developer’s with few hints on how to, concretely, implement such a solver. The only correctness proof [5] we are aware of stays also at an informal level, and hardly helps gaining some intuition about the key principles underlying the correctness of the algorithm. The only formally verified artifact that we know

of and that uses this technique is a (prototypical) static analyzer [4] precursor of the C Verasco analyzer [12]. There, the fixpoint computation is untrusted and validated *a posteriori* to be a post-fixpoint. But post-mortem fixpoint validation does not help understanding the sophisticated mechanics of the solvers.

In this work, we study the foundations of the fixpoint solving technique proposed by Bourdoncle [5], and conduct a formal correctness proof of both the iterative and recursive strategies. We apply our formalization work to the context of verified compilation, and implement both solvers within the CompCert compiler. We demonstrate the practicality of our verified algorithms with an empirical validation of their efficiency and precision, when used on a various collection of realistic dataflow analyses used for C program optimization and compilation. More precisely, we make the following contributions:

- We formalize the notion of weak topological ordering over the nodes of a graph, as well as two dataflow solver algorithms, following the informal descriptions given in the literature. Each solver implements a different solving strategy over an input WTO. The first algorithm implements the iterative strategy, the second implements the recursive strategy.
- We formally prove the correctness of both dataflow algorithms: they compute a post-fixpoint of a set of dataflow equations. Our proofs are generic in the WTO used by each of the strategy, and support both forward and backward dataflow analyses.
- Our formalization is done within the Coq proof assistant, and integrates to the reference, formally verified C compiler CompCert. Our algorithms are implemented in Coq, and then extractable to OCaml.
- We evaluate empirically the two formally verified solvers on the whole range of dataflow analyses provided by CompCert, using CompCert’s benchmark suite. We evaluate their performance in terms of efficiency and precision.

We present our contribution with a pen-and-paper formalism, and avoid to refer to Coq definitions as most as possible. Our Coq formal development and experimental validation setup are available online [15]. The full description of our experimental results is published as a companion experience report [16].

In this paper, we elide discussions regarding the termination of the algorithms, and simply assume they terminate. Our Coq formalization relies on fuel-bounded, mutually recursive functions, and are thus terminating.

The paper is structured as follows. Section 2 provides some technical background on dataflow analysis. In Section 3, we formalize the notion of a WTO, illustrating our definitions with examples, and we describe algorithmic ways to compute such an ordering. In Section 4, we explain the general architecture of a WTO-based dataflow solver, common to the two strategies, and the high-level structure of our correctness proof. Section 5 and Section 6 describe the algorithms for each of the solvers respectively, and the main technical lemmas we prove for the final correctness of the algorithms. In Section 7, we briefly comment on our Coq formal development and its integration within CompCert, and summarize our experimental results. Section 8 discusses related work. Section 9 concludes and discusses future improvements for this work.

2 Background on Dataflow Analysis

2.1 Dataflow Solvers

To perform dataflow analysis, programs are usually represented by their control-flow graph (CFG). A CFG $G = (N, n_e, E)$ is a graph where N is a set of nodes, E is a set of edges, and $n_e \in N$ is a distinguished entry node. The nodes correspond to program points and the edges describe the control flow between program points. We say that a node $s \in N$ is a *successor* of another node $n \in N$ if $(n, s) \in E$. The set of the successors of n in G is defined as the set $\text{succ}_G(n) = \{s \in N \mid (n, s) \in E\}$. Similarly, the set of the predecessors of s in G is defined as $\text{pred}_G(s) = \{n \in N \mid (n, s) \in E\}$.

The goal of dataflow analysis is to compute the most precise information possible at each program point *i.e.*, each node of the CFG. Information is usually represented by abstract values in a *semi-lattice* $(L, \sqsubseteq, \sqcup, \perp)$. Moreover, we are given a *transfer function* $T : N \times L \rightarrow L$, indexed by a node in the graph, that allows us to compute information at the output of the node given the information at its input. Namely, if $x \in L$ is the information at the input of n , then $T(n, x)$ is the information at the output.

A solution of the dataflow analysis problem can be represented as a mapping A from each node in the graph to a value in L . To be correct, the abstract information given by the solution should properly reflect the program data flow: for every successor s of a node n , we need to have $T(n, A[n]) \sqsubseteq A[s]$: the information at the output of n should at least be included in the value computed for s . Hence, the dataflow analysis problem consists in solving an (in)equation system on the nodes of the graph, and one must find a post-fixpoint for that system [21].

The general way to solve such a system is to solve the equations iteratively. Basically, one can retrieve information from predecessors p of a node s and update $A[s]$ in consequence to solve the equations of the form $T(p, A[p]) \sqsubseteq A[s]$. However, doing so may break the validity of the other equations concerning $A[s]$, hence requiring iteration passes over the nodes of the graph to reach a fixpoint.

If one iterates naively without minding the structure of the CFG, the convergence is likely to be very slow: a large number of iterations will have to be done before reaching the fixpoint. To make the iterations more efficient, several strategies exist. Kildall [13] gave a well-known historical algorithm with worklists, which is very commonly used in practice to perform dataflow analysis. However, other iteration strategies have been developed. Bourdoncle [5,6] elaborated a method which relies on a pre-computation of the iteration order. This iteration order is represented by a *weak topological ordering* (WTO) of the nodes. Given a WTO, Bourdoncle describes two iteration strategies : the iterative strategy and the recursive one. In this paper, we give a formalization for the notion of WTO (see Section 3) and for Bourdoncle's iteration strategies (see Section 4).

2.2 CompCert Interface

In the formally verified compiler CompCert, optimizations rely on a dataflow analysis, and the solvers are formally specified with the Coq proof assistant.

They are mostly specified as described in Section 2.1. However, the abstract structures used in CompCert have slightly weaker properties than a usual semi-lattice. In this paper, we call such structures *pseudo-semi-lattices*. The difference with a semi-lattice is that the join operator $\sqcup$ does not necessarily return the *least* upper bound but only an upper bound.

Definition 1 (Pseudo-semi-lattice). *A pseudo-semi-lattice is a quadruple $(L, \sqsubseteq, \sqcup, \perp)$ where $(L, \sqsubseteq)$ is a partially ordered set, $\sqcup$ is an upper bound operator and $\perp$ is a least element.*

A dataflow solver in CompCert is a function *fixpoint* that takes as input a CFG $G = (N, n_e, E)$, a pseudo-semi-lattice abstract domain $\mathbb{L} = (L, \sqsubseteq, \sqcup, \perp)$, a transfer function $T : N \times L \rightarrow L$, and an entry abstract value $v_e \in L$. It computes a solution A of the dataflow equation system as a mapping $A : N \mapsto L$ associating an abstract value to each of the nodes of G . CompCert’s interface for dataflow solvers also includes a correctness criterion stating that the returned solution is a fixpoint for the dataflow equation system.

Definition (Correctness criterion). *Let $G = (N, n_e, E)$ be a CFG. Assume $\text{fixpoint}(G, \mathbb{L}, T, v_e) = A$. The function *fixpoint* is said to be correct when, for all $n \in N$ reachable from n_e in G and $s \in \text{succ}_G(n)$, we have $T(n, A[n]) \sqsubseteq A[s]$.*

All of CompCert’s analyses rely on this dataflow solver interface, and implement it with Kildall’s historical algorithm [13]. In this work, we develop two new implementations for this interface in place of Kildall’s algorithm, namely the iterative and recursive strategies of Bourdoncle. The fact that all the algorithms implement the same interface saves us from modifying the rest of CompCert’s Coq development when replacing Kildall’s solver with Bourdoncle’s one in the compilation chain. It also makes it easy to compare both approaches experimentally in terms of efficiency and precision of the computed solution.

3 Weak Topological Ordering

The two dataflow solvers, proposed by Bourdoncle [5] and that we formalise in this paper, rely on a pre-computed data-structure. This data-structure guides the iteration strategy of the solvers, *i.e.*, the order in which the nodes of the CFG are processed in order to reach a fixpoint for the dataflow equations. This data-structure is called a *weak topological ordering*. The intuition that one usually has is that it represents, more or less faithfully, the loop structure of the CFG.

This section formally defines weak topological orderings. We illustrate our definitions with concrete examples, showing that a WTO does not necessarily reflect the loop nesting structure of the CFG. We comment on three methods to compute such an ordering from a graph. We finally describe our implementation, which we formally validate a posteriori using a formally verified validator.

3.1 Weak Topological Order

The notion of a *weak topological ordering* (WTO) is informally explained by Bourdoncle [5]. Here, we present our formal definition of a WTO. Let $G = (N, n_e, E)$ be a graph. A WTO is a data-structure containing nodes in N , arranged in a hierarchical way, called a *hierarchical order*. A WTO is then defined as a hierarchical ordering satisfying some well-formedness properties.

Definition 2 (Hierarchical Ordering). *A hierarchical ordering $\mathcal{H}$ is a list of node nestings containing no duplicate node. Each node nesting in the list is defined inductively as follows:*

$$\begin{array}{ll} n & \in N \quad \text{graph nodes} \\ e ::= \langle n \rangle & \text{vertex with single node } n \\ & | n @ e^* \quad \text{component with head node } n \text{ and node nestings list } e^* \end{array}$$

We emphasize that a hierarchical ordering is a list of such node nestings, rather than a node nesting itself: as will become clearer along the paper, it is sometimes better to avoid (and sometimes simply impossible) gathering all node nestings in a top-level node nesting. The fact that components are defined by both a head and a node nesting instead of just a node nesting disallows having two nested components opened at the same node, in accordance with Bourdoncle's definition of a hierarchical ordering [5].

Not all hierarchical orderings are valid WTO. We turn now to explaining and formalizing the required properties. The main property is a coherence criterion on the structure of the components of the hierarchical ordering. This criterion is captured through the notions of *parent head* and *feedback edge*.

In a hierarchical ordering, a parent head of a node is simply the head of any component enclosing that node.

Definition 3 (Parent head). *Let e be a node nesting. We say that v is a parent head of u in e if v is the head of a component of e containing u . Formally:*

- Node h is a parent head of h in $h @ l$
- If u belongs to l , then h is a parent head of u in $h @ l$
- If v is a parent head of u in some node nesting of l , then v is a parent head of u in $h @ l$

In a hierarchical ordering $\mathcal{H}$, v is a parent head of u if v is a parent head of u in some node nesting of $\mathcal{H}$.

The notion of feedback edge is more involved formally speaking, although it keeps relatively intuitive. It is defined relatively to a hierarchical ordering, and as such, it shouldn't be confused with the notion of a back-edge in a CFG¹. Indeed, a hierarchical ordering has a priori no relation to dominance. Meanwhile, feedback edges are defined with respect to the order in which the nodes appear in the hierarchical ordering. To define them, we rely on a partial order on the nodes of the graph, induced by a hierarchical ordering $\mathcal{H}$, written $\preceq_{\mathcal{H}}$.

¹ A back-edge in a graph is defined as an edge whose target dominates its source [1].

Definition 4 (Feedback edge). Let $G = (N, n_e, E)$ be a graph, and let $\mathcal{H}$ be a hierarchical ordering on G . An edge $(u, v) \in E$ is a feedback edge iff $v \preceq_{\mathcal{H}} u$.

We turn now to defining formally the partial order $\preceq_{\mathcal{H}}$. It suffices to define $x \preceq_{\mathcal{H}} y$ if and only if node x appears before y in the flattened version of $\mathcal{H}$. The *flattening* of a hierarchical ordering is the (unique) list containing exactly the same nodes, and that preserves the occurrence order of the nodes.

Definition 5 (Flattening of a hierarchical ordering). Let $\mathcal{H}$ be a hierarchical ordering. Its flattening, written $L(\mathcal{H})$, is defined by the following mutually recursive definitions, where *concat* and *map* are the usual operations on lists.

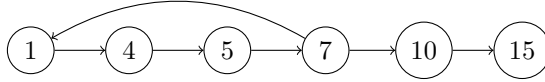
$$\begin{aligned} L(e^*) &= \text{concat}(\text{map}(\text{flatten}, e^*)) & \text{flatten}(\langle n \rangle) &= [n] \\ & & \text{flatten}(h @ e^*) &= h :: L(e^*) \end{aligned}$$

Definition 6 (Induced order of a hierarchical ordering). Let $\mathcal{H}$ be a hierarchical ordering, and $L(\mathcal{H})$ be its flattening. The induced order of $\mathcal{H}$, written $\preceq_{\mathcal{H}}$, is defined as the list order $\preceq_{L(\mathcal{H})}$ of $L(\mathcal{H})$, where for any list l of nodes:

$$\begin{array}{ccc} \frac{u \in l}{u \preceq_l u} & \frac{v \in m}{u \preceq_{u::m} v} & \frac{u \preceq_m v}{u \preceq_{x::m} v} \end{array}$$

The proof that $\preceq_{\mathcal{H}}$ is reflexive, antisymmetric and transitive is straightforward. $\preceq_{\mathcal{H}}$ is a total order on the nodes in $L(\mathcal{H})$.

Example 1. Let $G = (N, 1, E)$ be a graph defined by $N = \{1, 4, 5, 7, 10, 15\}$ and $E = \{(1, 4), (4, 5), (5, 7), (7, 1), (7, 10), (10, 15)\}$, and represented below



A first hierarchical ordering is $\mathcal{H}_1 = [1 @ [\langle 4 \rangle, \langle 5 \rangle, \langle 7 \rangle], \langle 10 \rangle, \langle 15 \rangle]$. The induced order of $\mathcal{H}_1$ is $[1, 4, 5, 7, 10, 15]$. Thus, for $\mathcal{H}_1$, $(7, 1)$ is the unique feedback edge. Another hierarchical ordering is $\mathcal{H}_2 = [\langle 5 \rangle, 7 @ [\langle 1 \rangle, 15 @ [\langle 10 \rangle]], \langle 4 \rangle]$. The induced order of $\mathcal{H}_2$ is $[5, 7, 1, 15, 10, 4]$ which has two feedback edges, $(4, 5)$ and $(10, 15)$.

In Example 1, while $\mathcal{H}_2$ is a hierarchical ordering, it does not constitute a valid basis to run an iterative dataflow solving algorithm. An important structural aspect is wrong: the feedback edge $(4, 5)$ has its source and target nodes split apart in two disjoint nestings. This is what forbids $\mathcal{H}_2$ from being a WTO.

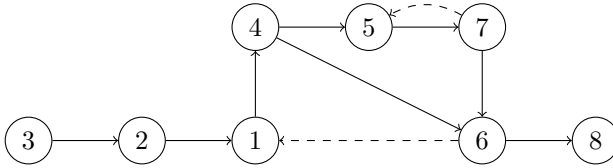
Definition 7 (Weak Topological Ordering (WTO)). Let $G = (N, n_e, E)$ be a graph. A weak topological ordering is a hierarchical ordering $\mathcal{W}$ on G satisfying the two following properties:

(Node completeness) All nodes of G reachable from n_e belong to $L(\mathcal{W})$

(Feedback-to-parent-head) For all feedback edge $(u, v) \in E$ of $\mathcal{W}$, v is a parent head of u .

In Example 1, both $\mathcal{H}_1$ and $\mathcal{H}_2$ are node-complete, but only $\mathcal{H}_1$ is satisfying the feedback-to-parent-head property, and $\mathcal{H}_2$ does not.

Example 2. We illustrate our definitions with a slightly more involved example.



The hierarchical ordering $\mathcal{W} = [\langle 3 \rangle, \langle 2 \rangle, 1 @ [\langle 4 \rangle, 5 @ [\langle 7 \rangle, \langle 6 \rangle], \langle 8 \rangle]]$ is indeed a WTO, with induced order $[3, 2, 1, 4, 5, 7, 6, 8]$. All nodes are reachable from the entry node 3 of the graph, and they are all included in $L(\mathcal{W})$. Edges (7, 5) and (6, 1) are the feedback edges (dashed) for $\mathcal{W}$. Since 5 is a parent head of 7 and 1 is a parent head of 6, then $\mathcal{W}$ is indeed a WTO. This example WTO is relatively intuitive, as it matches the loop structure of the graph: components correspond to the loops, and feedback edges point to a loop header.

But it is important to notice that this structural matching is not required for a hierarchical ordering to be a WTO. For instance, the following (degenerated) hierarchical ordering $[1@2@3@4@5@[6@[7@[8@[\]]]]]]$ is indeed a WTO.

Our specification of a WTO (Definition 7) is sufficient to prove the correctness of both dataflow solver algorithms that we present in Section 4, which adapts well to either forward dataflow solving, or backward solving on a reverse CFG. In our formalization, to accommodate backward analyses, we additionally make sure that the WTO contains all exit nodes of the CFG.

Yet, from a performance perspective, the particular choice of a WTO is crucial. For instance, the degenerated WTO `[1@2@[3@[\dots, 7@[8@[]]]]]` from Example 2, would imply a high number of iterations over large collections of nodes to reaching a fixpoint. Intuitively, WTO-based dataflow solvers perform better when the WTO is faithful to the CFG structure: typically, feedback edges link each end of loop body to the header of the corresponding loop. Indeed, reflecting the structure of the CFG is likely to reflect the underlying dependencies between dataflow equations. It is, however, difficult to make this observation more than a simple intuition, generally speaking.

3.2 Building a WTO

Previous examples and discussions show that a single graph admits in general several WTO. There exist indeed several algorithms to construct a WTO.

Enumeration Method. This is probably the simplest algorithm. It consists in listing all the nodes of the graph in an arbitrary order, which then serves as the induced order of the hierarchical ordering under construction. Following this order, the list of node nestings is then defined recursively by listing all the

nodes as vertices with a single node, opening a component as soon as necessary – *i.e.*, when an encountered node is pointed to by a feedback edge – and finally, closing all components at the very end. For a node list l without duplicates, the algorithm is a simple recursive function $\mathcal{W}(G, l)$

$$\begin{aligned} \mathcal{W}(G, []) &= [] \\ \mathcal{W}(G, x :: m) &= \langle x \rangle :: \mathcal{W}(m) && \text{if there is no } (p, x) \in E, \text{ with } p \geq x \\ &= x @ \mathcal{W}(m) && \text{if there is } (p, x) \in E \text{ with } p \geq x \end{aligned}$$

Example 3. In Example 2, when listing the nodes by number, the WTO computed by the enumeration method is $[\langle 1 \rangle @ [2 @ [\langle 3 \rangle, \langle 4 \rangle, 5 @ [\langle 6 \rangle, \langle 7 \rangle, \langle 8 \rangle]]]]$.

In this work, we implemented the above algorithm and proved its correctness, with respect to Definition 7: the algorithm does compute a hierarchical ordering that is a WTO. Our correctness proof consists in formalizing the following reasoning. First, as all nodes of the graph are listed, regardless of their reachability from the entry node, the resulting ordering is node-complete. Moreover, when a feedback edge (u, v) is encountered, *i.e.*, when $v \leq u$, then v is indeed a parent head of u , since the just opened component with head v contains all the nodes that are still to be processed recursively in the list, including u .

To us, the Enumeration method is essentially a proof of concept for our formal specification of a WTO. But the computed WTO does not a priori reflect the structure of the CFG, and results in a considerably slow convergence of WTO-based solvers. Bourdoncle describes a specific enumeration method where the nodes are discovered in a depth-first search (DFS) of the graph instead of an arbitrary order [5]. Instead, we implemented another method that gives a clearly better WTO than the Enumeration method, that we present next.

Strongly Connected Components Method. This is probably the method that is the most frequently employed to build a WTO. The algorithm [5] decomposes the graph in strongly connected components and subcomponents, using Tarjan’s algorithm [25]. It extends Tarjan’s algorithm to construct the WTO on the fly during the SCC discovery, by recursively applying the algorithm on discovered SCC, once the root of an SCC and all the edges pointing to that root are virtually disconnected. The SCC (sub-)components correspond to the WTO (sub-)components. The algorithm starts the exploration of the graph from its entry node, and discovers the (sub-)components in a topological order, as in Tarjan’s algorithm. The feedback edges in the resulting WTO are those edges pointing to the root of an SCC component.

Example 4. The SCC of the graph from Example 2 are, in their topological order, $\{3\}$, $\{2\}$, $\{1, 4, 5, 7, 6\}$ which has one subcomponent $\{5, 7\}$, and finally $\{8\}$. The discovered back edges are $(7, 5)$ and $(6, 1)$. Consequently, the WTO computed using the SCC method is $[\langle 3 \rangle, \langle 2 \rangle, 1 @ [\langle 4 \rangle, 5 @ [\langle 7 \rangle], \langle 6 \rangle], \langle 8 \rangle]$.

The SCC method has a time complexity in $O(\delta|E|)$ and provides a WTO which generally reflects well the program structure. In our development, we reused an implementation of the SCC method provided by the OCamlGraph library [9]. An informal proof of the correctness of this method is presented in Bourdoncle’s manuscript [6]. Although formalizing the proof could certainly be feasible, relying on previous work mechanizing the correctness proofs of Tarjan’s algorithm [8], such a proof is quite orthogonal and out of the scope of the present work, which focuses primarily on the correctness of dataflow solvers algorithms.

Rather than formally verifying the SCC method for building a WTO, we thus implement a formally verified validator in Coq, which checks, a posteriori, that the untrusted computed WTO effectively meets its specification. This does not lower the formal guarantee of the final implementation: the validator is formally proved in Coq to only accept hierarchical orderings that are WTO, with respect to Definition 7. Furthermore, in practice, our validator never rejects a WTO that has been computed with the SCC method.

Our validator for a WTO candidate $\mathcal{W}$ takes as inputs the candidate $\mathcal{W}$ and the CFG for which $\mathcal{W}$ is supposed to be a WTO. In a first step, from $\mathcal{W}$, we build a map $\mathcal{O}(\mathcal{W})$, associating to each node its index in the flattening $L(\mathcal{W})$. During the computation of the map, we check for potential duplicates. When the map is computed, we hence know that $\mathcal{W}$ is a hierarchical ordering for G . In a second step, we validate the node completeness property of Definition 7. We check that the entry node is in $L(\mathcal{W})$ and then check that $\mathcal{W}$ is closed by the successor relation of the graph G . It follows that all reachable nodes from the entry node are in $\mathcal{W}$. Finally, we validate the feedback-to-parent-head property of Definition 7. Here, we rely on the map $\mathcal{O}(\mathcal{W})$ to determine whether an edge (u, v) is a feedback edge or not, for $\mathcal{W}$, by comparing the indices $\mathcal{O}(\mathcal{W})[u]$ and $\mathcal{O}(\mathcal{W})[v]$ in the map. We then check, straightforwardly, for each such feedback edge (n, s) , that s is a parent head of n by looking in enclosing components.

The overall complexity of the validator is $O(|E| \log(|N|) + |N|(\log|N| + \text{feedback}(G)))$ where $\text{feedback}(G)$ is the number of feedback edges in G . For realistic CFG, we have $|E| = O(|N|)$: instructions almost never have more than 2 successors. Therefore we can broadly approximate the complexity of the validator to $O(|N|(\log|N| + \text{feedback}(G)))$. This is confirmed experimentally: we could observe that the ratio between validation time for a CFG G with n nodes and $n(\log n + \text{feedback}(G))$ asymptotically tends to a constant.

4 General Framework of WTO-based Dataflow Solvers

Equipped with a WTO, we describe here the two dataflow solvers algorithms that we have formalized in Coq. For the sake of the presentation, we focus on describing the algorithm in pseudo-code, and highlighting the important parts of their correctness proof.

The first algorithm implements the so-called iterative strategy, while the second implements the recursive strategy [5]. While being significantly different one from the other, both algorithms fit a common, global framework, which

Algorithm 1 Common algorithmic part of WTO-based dataflow solvers

Inputs: $G = (N, n_e, E)$, $\mathbb{L} = (L, \sqsubseteq, \sqcup, \perp)$, $T : N \times L \rightarrow L$, $v_e \in L$, $\mathcal{W}$
Output: A fixpoint solution $A : N \mapsto L$

```

1:  $S \leftarrow \emptyset$ 
2:  $A_{init} \leftarrow \{n_e \mapsto v_e\}$ 
3:  $A \leftarrow A_{init}$ 
4: for each node nesting  $e$  in  $\mathcal{W}$  do
5:   match  $e$  with
6:   |  $\langle n \rangle \Rightarrow A[n] \leftarrow \text{ANALYZENODE}(n)$ 
7:   |  $h@l \Rightarrow \text{STABILIZE}(h, l)$ 
8:    $S \leftarrow S \cup \{e\}$ 
9: return  $A$ 

10: function ANALYZENODE( $n$ )
11:    $v \leftarrow A_{init}[n]$ 
12:   for each  $p \in \text{pred}_G(n)$  do
13:     if  $v \neq T(p, A[p])$  then
14:        $v \leftarrow v \sqcup T(p, A[p])$ 
15:   return  $v$ 
    
```

reflects in the general architecture of our correctness proofs. We thus start by describing this global framework, as well as its key ingredients and intuitions.

WTO-based dataflow solvers follow CompCert’s interface for *fixpoint* described in Section 2.2, with the exception that they take a WTO $\mathcal{W}$ as an extra input. The resulting formal correctness theorem for WTO-based solvers becomes:

Theorem 1 (Fixpoint partial correctness). *Let $G = (N, n_e, E)$ be a CFG. Assume $\text{fixpoint}(G, \mathbb{L}, T, v_e, \mathcal{W}) = A$. For all $n \in N$ reachable from n_e in G and all $s \in \text{succ}_G(n)$, we have $T(n, A[n]) \sqsubseteq A[s]$.*

Notice that we only consider a partial correctness result, and assume the algorithm is indeed terminating. Formally proving its termination is outside of the scope of our contribution. In our Coq formalization, we resort to a fuel argument to ensure termination: the algorithm takes an extra parameter (a natural number) arbitrarily bounding the number of mutual recursive calls, and returns an optional result A that is defined only when enough fuel is provided.

4.1 Common Algorithmic Part

The pseudo-code for *fixpoint* is given in Algorithm 1. The resulting mapping $A : N \mapsto L$, initialized with v_e for the entry node n_e , is updated along the algorithm. It represents the solution of the equation system, described by T , computed so far. Then, the algorithm traverses $\mathcal{W}$, visiting each node nesting in order, and updating A accordingly. When visiting a vertex $\langle n \rangle$, we simply process the corresponding node n with function ANALYZENODE: in the current context, the abstract information for that node is updated by joining the current value with the abstract value coming from each of its predecessors. Visiting a component $h@l$ consists in *stabilizing* that component, using the STABILIZE procedure. Essentially, when STABILIZE returns, the dataflow equations should hold for all the CFG nodes inside $h@l$. Both algorithms differ in the way they stabilize a component: the iterative strategy stabilizes a top-level node nesting

without taking into account the structure of its subcomponents, while the recursive strategy recursively stabilize subcomponents. For the moment, we keep the definition of STABILIZE abstract.

Handling Backward Analyses The algorithm described here uses a CFG with a single entry node. But this implies some adaptations to support backward analyses, on a reversed CFG: we cannot assume that the CFG has a single exit point. To keep the presentation lighter, we abstract from those technical discrepancies between a forward and a backward dataflow solver algorithm. Essentially, the algorithm for forward analyses, as well as the main proof arguments, can largely be reused as is to implement a backward dataflow solver.

4.2 Common Proof Architecture

Both dataflow solvers algorithms follow a similar high-level proof architecture. We describe it now. Considering that each node nesting is visited one after the other in the main loop of the algorithm, dataflow equations should hold for the nodes belonging to the node nestings that were processed previously. We introduce a ghost variable S in our algorithm, to keep track of the set of the previously processed node nestings. We use this ghost variable to formulate the main correctness invariant of the algorithm. This invariant basically means that dataflow equations hold on all CFG nodes in (a node nesting of) S : $\forall e \in S, \forall n \in e, \forall p \in \text{pred}(n), T(p, A[p]) \sqsubseteq A[n]$. Here $n \in e$ means, by abuse of notation, that n is in $\text{flatten}(e)$. We say that e contains a node n , or n belongs to e .

For a set of node nestings S , we write $\text{Nodes}(S)$ the set of nodes belonging to some nesting in S . Formally, $\text{Nodes}(S) = \{n \in N \mid \exists e \in S, n \in e\}$. We additionally define the predicate $\text{Valid}(A, n) \triangleq \forall p \in \text{pred}(n), T(p, A[p]) \sqsubseteq A[n]$, expressing that, for the current value of A , the dataflow equations hold at node n . Using these notations, the main correctness invariant can thus be rephrased as $\forall s \in \text{Nodes}(S), \text{Valid}(A, s)$. For both strategy algorithm, we prove that this invariant is preserved through the main loop of Algorithm 1.

Lemma 1 (Correctness invariant of S). *After any iteration of the top-level loop of Algorithm 1, we have $\forall s \in \text{Nodes}(S), \text{Valid}(A, s)$.*

Naturally, the proof of the above lemma is specific to each of the strategy algorithms, which defines its own version of STABILIZE procedure (see Sections 5 and 6). But this lemma is enough to prove the above correctness theorem. Indeed, at the end of that top-level loop, all the node nestings in $\mathcal{W}$ are in S . By definition of a WTO, the invariant thus holds on all the nodes of G that are reachable from the entry node n_e . Furthermore, thanks to the fact that the WTO is closed by the successor relation, the invariant phrased on predecessors, is equivalent to the above correctness theorem phrased on successors.

An important point to note in Algorithm 1 is that the nestings in $\mathcal{W}$ cannot be processed in any order: they must be processed in the order in which they appear in the WTO. Indeed, in a WTO, a feedback edge can only link two nodes in the same top-level nesting thanks to the feedback-to-parent-head property. Thus, the

predecessors of a node n either appear before n or are in the same node nesting as n . As a result, if the nestings are processed in the correct order, stabilized nodes can only have stabilized nodes as predecessors. This guarantees that the equations on stabilized nodes are preserved when stabilizing new components. However, if the nestings are processed in the wrong order, some stabilized node n might have not yet stabilized nodes as predecessors (they will appear before n in the WTO). The equations at already stabilized nodes might then be broken by further processings, which leads to an incorrect result. We formalize this observation by the following lemma, which we prove as invariant of the top-level loop of the algorithm. The proof essentially relies on the definition of a WTO.

Lemma 2 (Predecessor-closure of S). *At any point of the execution of Algorithm 1, the set $\text{Nodes}(S)$ is closed by the predecessor relation: for all $n \in \text{Nodes}(S)$, and all $p \in \text{pred}(n)$, we have $p \in \text{Nodes}(S)$.*

At the core of both dataflow algorithms is the function `ANALYZENODE`. We prove a basic lemma stating that it returns an abstract value greater than the current values at all its predecessors. The proof is straightforward.

Lemma 3 (Correctness of `ANALYZENODE`). *Let $n, p \in N$ be two nodes of the CFG. If $p \in \text{pred}(n)$, then $T(p, A[p]) \sqsubseteq \text{ANALYZENODE}(n)$.*

5 WTO-Iterative Dataflow Solver

5.1 The Algorithm

What distinguishes the iterative strategy from the recursive one is the implementation of `STABILIZE( $h, l$ )`. The iterative strategy proceeds as follows. An iteration consists in retrieving information from the predecessors of each node n in $h :: L(l)$. Iterations are performed until a fixpoint is reached on these nodes. To detect the stabilization of the component after an iteration (*i.e.*, to detect that the fixpoint has been reached), Bourdoncle has shown that it suffices to check that the equations hold at h and the heads of the subcomponents of l [5]. However we must ensure that we analyze each node of the component at least once before declaring the component as stabilized. Indeed, non-head nodes inside the component might have predecessors in nestings previously processed. We must analyze these nodes to retrieve the final information computed at their predecessors, otherwise the correctness of the analysis is not guaranteed.

`STABILIZE( $h, l$ )` can then be implemented as follows. At the first iteration, for each node n in $h :: L(l)$, retrieve information from all its predecessors. For next iterations, proceed each node n in $h :: L(l)$ as follows. If $n = h$ or n is the head of a subcomponent in l , retrieve information from the predecessors of n . If the equations hold at n , then do nothing. Otherwise, the rest of the component has to be entirely analyzed again, like at the first iteration. If n is not a head and all the previous checks at heads succeeded, then do nothing. The process stops once all the checks at heads have succeeded at the last iteration.

Algorithm 2 Bourdoncle's iterative strategy**Inputs:** $G = (N, n_e, E)$, $\mathbb{L} = (L, \sqsubseteq, \sqcup, \perp)$, $T : N \times L \rightarrow L$, $v_e \in L$, $\mathcal{W}$ **Output:** A fixpoint solution $A : N \mapsto L$

```

1:  $S \leftarrow \emptyset$ 
2:  $A_{init} \leftarrow \{n_e \mapsto v_e\}$ 
3:  $A \leftarrow A_{init}$ 
4: for each node nesting  $e \in \mathcal{W}$  do
5:   match  $e$  with
6:   |  $\langle n \rangle \Rightarrow A[n] \leftarrow \text{ANALYZENODE}(n)$ 
7:   |  $h@l \Rightarrow \text{STABILIZE}(h, l)$ 
8:    $S \leftarrow S \cup \{e\}$ 
9: return  $A$ 
10: function  $\text{ANALYZENODE}(n)$ 
11:    $v \leftarrow A_{init}[n]$ 
12:   for each  $p \in \text{pred}_G(n)$  do
13:     if  $v \neq T(p, A[p])$  then
14:        $v \leftarrow v \sqcup T(p, A[p])$ 
15:   return  $v$ 
16: procedure  $\text{ANALYZENESTING}(e)$ 
17:   match  $e$  with
18:   |  $\langle n \rangle \Rightarrow$ 
19:      $A[n] \leftarrow \text{ANALYZENODE}(n)$ 
20:   |  $h@l \Rightarrow$ 
21:      $A[h] \leftarrow \text{ANALYZENODE}(h)$ 
22:     for each node nesting  $e' \in l$  do
23:        $\text{ANALYZENESTING}(e')$ 
24: function  $\text{CHECKCOMPONENT}(h, l)$ 
25:    $v_h \leftarrow A[h] \sqcup \text{ANALYZENODE}(h)$ 
26:   if  $v_h = A[h]$  then
27:     return  $\text{CHECKNESTINGS}(l)$ 
28:   else
29:      $A[h] \leftarrow v_h$ 
30:     for each node nesting  $e \in l$  do
31:        $\text{ANALYZENESTING}(e)$ 
32:   return true
33: function  $\text{CHECKNESTING}(e)$ 
34:   match  $e$  with
35:   |  $\langle n \rangle \Rightarrow$  return false
36:   |  $h@l \Rightarrow$ 
37:     return  $\text{CHECKCOMPONENT}(h, l)$ 
38: function  $\text{CHECKNESTINGS}(l)$ 
39:    $m \leftarrow \text{false}$ 
40:   for each node nesting  $e \in l$  do
41:     if  $m$  then
42:        $\text{ANALYZENESTING}(e)$ 
43:     else
44:        $m \leftarrow \text{CHECKNESTING}(e)$ 
45:   return  $m$ 
46: procedure  $\text{STABILIZE}(h, l)$ 
47:    $\text{ANALYZENESTING}(h@l)$ 
48:    $m \leftarrow \text{false}$ 
49:   repeat
50:      $m \leftarrow \text{CHECKCOMPONENT}(h, l)$ 
51:   until  $m = \text{false}$ 

```

The pseudo-code of Bourdoncle's iterative strategy is given in Algorithm 2. The auxiliary function ANALYZENODE returns information retrieved from the predecessors of a node. The first iteration logic is implemented by the ANALYZENESTING function. This function is called at the beginning of STABILIZE and when a check fails in the subsequent iterations: the remaining nodes of the component have to be re-analyzed. ANALYZENESTING updates the abstract values of the nodes in a nesting with the information retrieved.

The mutually recursive functions CHECKCOMPONENT , CHECKNESTING and CHECKNESTINGS implement subsequent iterations. They check for stabilization at heads and analyze remaining nestings with ANALYZENESTING once a check fails. The results of the checks are propagated thanks to the boolean m : while it is **false**, abstract values at heads have not been modified (the checks succeeded). CHECKNESTING checks an individual nesting. The check succeeds imme-

diately for a vertex (it is not a head). The check for a component is done with `CHECKCOMPONENT`. `CHECKCOMPONENT` checks the head of a component and recursively checks its sub-nestings with `CHECKNESTINGS`. If the check fails, sub-nestings are re-analyzed with `ANALYZENESTING`. `CHECKNESTINGS` checks a list of nestings. If a check fails at some point ($m = \text{true}$), remaining nestings are re-analyzed with `ANALYZENESTING`.

5.2 Proof of Correctness

The correctness proof of Algorithm 2 follows the general architecture described earlier. We thus describe now the salient point of the proof of Lemma 1. We split the proof in two main steps. First, we prove that stabilizing a given node nesting does not break the validity of the equations for nodes in the nestings that were previously stabilized (Lemma 4). Then, we prove that stabilizing a given nesting e indeed ensures the validity of the equations for the nodes of e (Lemma 6).

Notations. In the following, we make use of the following notation. We write $S_0, A_0 \xrightarrow{F(x)} S_1, A_1$ for the property “if $S = S_0$ and $A = A_0$ before a call to $F(x)$, then $S = S_1$ and $A = A_1$ after the call.”. For example, if $S = S_0$, $A = A_0$ before a call to `STABILIZE( $h, l$ )` and $S = S_1$, $A = A_1$ when the function returns, then we have $S_0, A_0 \xrightarrow{\text{STABILIZE}(h, l)} S_1, A_1$.

We now present the main technical lemmas, and give an overview of their proofs by giving the key auxiliary lemmas involved therein.

STABILIZE preserves validity. As explained above, we first prove that for nodes already processed by the procedure, and hence belonging to S , the validity of their dataflow equation is preserved.

Lemma 4 (STABILIZE preserves validity).

Suppose $S_0, A_0 \xrightarrow{\text{STABILIZE}(h, l)} S_1, A_1$ and that for all $n \in h@l$, $n \notin \text{Nodes}(S_0)$. Then for all $s \in \text{Nodes}(S_0)$, if $\text{Valid}(A_0, s)$ then $\text{Valid}(A_1, s)$.

To prove Lemma 4, we rely on an auxiliary lemma (Lemma 5) which states that `ANALYZENESTING` preserves the validity of equations for nodes that do not have predecessors in the node nesting argument. We prove a similar result for `CHECKCOMPONENT`, that we elide here to keep the presentation light.

Lemma 5. *Assume that $S_0, A_0 \xrightarrow{\text{ANALYZENESTING}(e)} S_1, A_1$. Let $s \notin e$ having no predecessors in e . If $\text{Valid}(A_0, s)$, then $\text{Valid}(A_1, s)$.*

STABILIZE ensures validity. For the nodes analyzed through the procedure, we show that their dataflow equations become valid at the end of the process.

Lemma 6 (STABILIZE ensures validity). *Suppose $S_0, A_0 \xrightarrow{\text{STABILIZE}(h, l)} S_1, A_1$. Then for all s in $h@l$, we have $\text{Valid}(A_1, s)$.*

To prove Lemma 6, we split the reasoning in three steps. First, we show in Lemma 7 below that, after analyzing the nodes of a nesting with ANALYZENESTING, the equations hold on all the analyzed nodes that are not heads of sub-components in the WTO. Indeed, a non-head node n have no predecessors after it in the WTO (by the feedback-to-parent-head property of a WTO). Thus the values at the predecessors of n have not changed after we analyzed n . However we might have new information to retrieve at heads: they might have predecessors analyzed after them.

At this point in the proof, we need to distinguish nodes that are component heads from the ones that are not. To do this, we rely on the notion of parent head (Definition 3). In a node nesting e , s is the head of a sub-component of e if and only if s is a parent head of itself in e . In particular, if s is a parent head of some node n , then s is a parent head of itself.

Lemma 7. *Suppose $S_0, A_0 \xrightarrow{\text{ANALYZENESTING}(e)} S_1, A_1$. Then for all $s \in e$, if s is not a parent head of itself in e , then $\text{Valid}(A_1, s)$.*

Next, we prove in Lemma 8 below, that provided the equations hold on all non-head nodes of the component, they still hold after checking the component with CHECKCOMPONENT. This is true for reasons similar to the first part.

Lemma 8. *Suppose $S_0, A_0 \xrightarrow{\text{CHECKCOMPONENT}(h,l)} S_1, A_1$. Let s in $h@l$. If s is not a parent head of itself in $h@l$ and $\text{Valid}(A_0, s)$, then $\text{Valid}(A_1, s)$.*

Finally, we prove with Lemma 9 that, once CHECKCOMPONENT returns false, the equations hold on all the nodes of the component, including heads. This is true provided that they hold on non-head nodes before the final check. Indeed, if the checks at heads at line 26 succeed, then the equations hold on the heads.

Lemma 9. *Suppose $S_0, A_0 \xrightarrow{\text{CHECKCOMPONENT}(h,l)} S_1, A_1$. Assume further that for all s in $h@l$, if s is not a parent head of itself in $h@l$ then $\text{Valid}(A_0, s)$. If CHECKCOMPONENT returns false, then, for all $s \in h@l$, we have $\text{Valid}(A_1, s)$.*

6 WTO-Recursive Dataflow Solver

6.1 The Algorithm

The recursive strategy implements STABILIZE(h, l) differently from the iterative strategy. Here, stabilizing a component recursively stabilizes the encountered sub-components. Bourdoncle indicates that with this strategy, it would be sufficient to check that the equations hold at the head of the component [5] to detect the stabilization of a whole component. This simplifies the algorithm, the heads of sub-components do not need to be checked for stability, and the need to use functions CHECKNESTING, CHECKNESTINGS and CHECKCOMPONENT vanishes.

The pseudo-code of the recursive strategy is given in Algorithm 3. We explain now the functioning of STABILIZE(h, l). Here again, we distinguish the first

iteration on the component from the next iterations. For the first iteration, visit each node nesting in $\langle h \rangle :: l$. If it is a vertex $\langle n \rangle$, we simply analyze node n . Otherwise, visiting a component $h'@l'$ consists in stabilize it, recursively, by calling $\text{STABILIZE}(h', l')$. For the subsequent iterations on $h@l$, retrieve information from the predecessors of h and join it with the current value available for h . If the result is not greater than $A[h]$, then stop. Otherwise, analyze each nesting of l like at the first iteration.

Algorithm 3 Bourdoncle’s recursive strategy

Inputs: $G = (N, n_e, E)$, $\mathbb{L} = (L, \sqsubseteq, \sqcup, \perp)$, $T : N \times L \rightarrow L$, $v_e \in L$, $\mathcal{W}$

Output: A fixpoint solution $A : N \mapsto L$

<pre> 1: $S \leftarrow \emptyset$ 2: $A_{init} \leftarrow \{n_e \mapsto v_e\}$ 3: $A \leftarrow A_{init}$ 4: for each node nesting $e \in \mathcal{W}$ do 5: match e with 6: $\langle n \rangle \Rightarrow$ 7: $A[n] \leftarrow \text{ANALYZENODE}(n)$ 8: $h@l \Rightarrow \text{STABILIZE}(h, l, \text{false})$ 9: $S \leftarrow S \cup \{e\}$ 10: return A 11: function $\text{ANALYZENODE}(n)$ 12: $v \leftarrow A_{init}[n]$ 13: for each $p \in \text{pred}(n)$ do 14: if $v \neq T(p, A[p])$ then 15: $v \leftarrow v \sqcup T(p, A[p])$ 16: return v </pre>	<pre> 17: procedure $\text{ANALYZENESTING}(e)$ 18: match e with 19: $\langle n \rangle \Rightarrow$ 20: $A[n] \leftarrow \text{ANALYZENODE}(n)$ 21: $h@l \Rightarrow \text{STABILIZE}(h, l, \text{false})$ 22: procedure $\text{STABILIZE}(h, l, \text{stop})$ 23: $v_h \leftarrow A[h] \sqcup \text{ANALYZENODE}(h)$ 24: if $v_h \neq A[h]$ or not stop then 25: $A[h] \leftarrow v_h$ 26: for each node nesting $e \in l$ do 27: $\text{ANALYZENESTING}(e)$ 28: $\text{STABILIZE}(h, l, \text{true})$ </pre>

In Algorithm 3, we provide an extra boolean parameter to STABILIZE , stop , that guarantees that the first iteration is always done and that the nodes of the component are analyzed at least once. The initial call to STABILIZE is made with $\text{stop} = \text{false}$ while the recursive calls are made with $\text{stop} = \text{true}$. The function only returns when stop is true².

6.2 Proof of Correctness

The correctness proof for Algorithm 3 follows the general proof architecture explained in Section 4. We split the proof of the correctness invariant preservation (Lemma 1) in a way that is similar to the proof of the iterative dataflow algorithm. We first prove that STABILIZE preserves the validity of equation for the nodes in the already stabilized node nestings (Lemma 10). We then prove

² Introducing this parameter simplifies the implementation and the correctness proof.

that STABILIZE effectively ensures the validity of equations for the nodes of the component under stabilization (Lemma 11).

Lemma 10 (STABILIZE preserves validity).

Suppose $S_0, A_0 \xrightarrow{\text{STABILIZE}(h, l, \text{stop})} S_1, A_1$ and for all $n \in h@l$, $n \notin \text{Nodes}(S_0)$. Then for all $s \in \text{Nodes}(S_0)$, if $\text{Valid}(A_0, s)$ then $\text{Valid}(A_1, s)$.

Proving Lemma 10 for Algorithm 3 is much simpler than for Algorithm 2, since we don't have to consider auxiliary functions that implement the stabilization check. Hence, we don't expand here on auxiliary lemmas.

Next, we prove with Lemma 11 that STABILIZE does indeed establish the validity of the equations for the nodes it analyzes recursively.

Lemma 11 (STABILIZE ensures validity). *Suppose $S_0, A_0 \xrightarrow{\text{STABILIZE}(h, l, \text{false})} S_1, A_1$. Then for all $s \in h@l$, $\text{Valid}(A_1, s)$.*

To prove Lemma 11, we proceed in a different way than previously. In particular, this proof has to be done by mutual recursion between ANALYZE_NESTING and STABILIZE. In our implementation, we add a natural *fuel*, p , as a parameter to both functions to ensure termination. This fuel integer allows us here to reason by strong induction on p to do the proofs. Consequently, we make that fuel parameter explicit in the following lemmas, and write $\text{ANALYZE\_NESTING}_p(e)$ and $\text{STABILIZE}_p(h, l, \text{stop})$ the calls with a fuel equal to p .

We organize the proof of Lemma 11 using three auxiliary properties, each of them indexed with a current level of fuel p :

$\mathcal{C}_{Ana}(p)$ states that after analyzing a node nesting e with $\text{ANALYZE\_NESTING}_p(e)$, the equations hold on all the nodes in e , including heads.

$\mathcal{C}_{Stab}(\text{true}, p)$ states that after calling $\text{STABILIZE}_p(h, l, \text{true})$ to stabilize a component, the equations hold on all nodes in $h@l$, provided they were holding on all the nodes in l .

$\mathcal{C}_{Stab}(\text{false}, p)$ states that after calling $\text{STABILIZE}_p(h, l, \text{false})$ to stabilize a component, the equations hold on all the nodes of $h@l$, without any extra prerequisite.

We do the proof along the following induction scheme. First, assuming that for all $k < p$ we have $\mathcal{C}_{Ana}(k)$, we prove that property $\mathcal{C}_{Stab}(\text{true}, p)$ holds. But, for any p , $\mathcal{C}_{Stab}(\text{true}, p)$ implies $\mathcal{C}_{Stab}(\text{false}, p + 1)$. Indeed, when executing a call $\text{STABILIZE}_{p+1}(h, l, \text{false})$, we call $\text{STABILIZE}_p(h, l, \text{true})$ after analyzing at least once all the nodes in l , thus solving their equations. As a result, assuming that for all $k < p$ (a fortiori for all $k < p - 1$) we have $\mathcal{C}_{Ana}(k)$, we can deduce that $\mathcal{C}_{Stab}(\text{false}, p)$. We can conclude the proof of $\mathcal{C}_{Ana}(p)$ by strong induction on p .

Another difference with the proof of Algorithm 2 is that, thanks to the recursive stabilizations of subcomponents, we are able to prove that, after one iteration on a component $h@l$, the algorithm keeps the equations valid on all the nodes of l rather than just on the non-head node of l . Indeed, the equations at the head h are the only ones that might not hold. Equations at the heads of the subcomponents, and thus all non-head nodes in subcomponents, have been solved recursively.

Lemma 12. $\forall p, (\forall k < p, \mathcal{C}_{Ana}(k)) \Rightarrow \mathcal{C}_{Stab}(p, true)$.

Let $p \in \mathbb{N}$, and suppose that for all $k < p$, if $S_0, A_0 \xrightarrow{\text{ANALYZE\_NESTING}_k(e)} S_1, A_1$, then for all $s \in e$, $Valid(A_1, s)$. Suppose also that $S_0, A_0 \xrightarrow{\text{STABILIZE}_p(h, l, true)} S_1, A_1$, and that for all $s \in l$, we have $Valid(A_0, s)$. Then for all $s \in h @ l$, we have $Valid(A_1, s)$.

Lemma 13. $\forall p, (\forall k < p, \mathcal{C}_{Ana}(k)) \Rightarrow \mathcal{C}_{Stab}(p, false)$.

Let $p \in \mathbb{N}$ and suppose that, for all $k < p$, if $S_0, A_0 \xrightarrow{\text{ANALYZE\_NESTING}_k(e)} S_1, A_1$, then for all $s \in e$, $Valid(A_1, s)$. Suppose also that $S_0, A_0 \xrightarrow{\text{STABILIZE}_p(h, l, false)} S_1, A_1$. Then for all $s \in h @ l$, we have $Valid(A_1, s)$.

Lemma 14 (ANALYZE_NESTING ensures validity). $\forall p, \mathcal{C}_{Ana}(p)$.

Let $p \in \mathbb{N}$. If $S_0, A_0 \xrightarrow{\text{ANALYZE\_NESTING}_p(e)} S_1, A_1$, then for all $s \in e$, $Valid(A_1, s)$.

7 Implementation and Experimental Evaluation

We now evaluate experimentally our verified dataflow solvers. Our first objective is to confirm experimentally that our verified and implemented algorithms have acceptable performances, both in terms of precision and efficiency. Our baseline is the generic dataflow solvers from CompCert’s distribution, implementing Kildall’s algorithm [13]. We do not expect to outperform CompCert’s dataflow solvers. Indeed, CompCert’s implementation of Kildall’s algorithm relies on well-chosen data structures, in particular to implement worklists. The latter are based on heap structures from CompCert library whose performances have already been carefully tuned, and trade-offs regarding compilation time and optimization power have already been worked out.

Our second objective is to better understand the practical differences between the solvers, by delineating their respective strengths and weaknesses observed in practice, in the realistic setting of a compiler toolchain like CompCert.

Below we provide a summary of our experimental results and conclusions. Our companion experiment report [16] further details experimental data and results, drawn from three representative dataflow analyses: (1) ValueAnalysis, a forward analysis starting from the CFG entry point, (2) Liveness analysis, a backward analysis starting from a overapproximated set of exit points, and (3) Allocation, a backward analysis where all nodes are considered as exit points.

7.1 Code Size of our Development

Our complete development includes Coq and OCaml code. Coq files contain specifications, programs and proofs. Coq programs correspond to both dataflow solvers, including their forward and backward versions, as well as a formally verified validator for our WTO computation. The WTO computation itself is hand-written in OCaml. It implements Tarjan’s SCC algorithm, and is mainly

Table 1: Size of our development (LoC), with regards to CompCert’s code base

	Coq Definitions	Coq Proofs	OCaml code	Total
WTO axiomatization	409	504	—	913
WTO validation	386	323	—	709
WTO computation	—	—	160	160
Iterative Bourdoncle	1399	2037	—	3436
Recursive Bourdoncle	946	1541	—	2487
Total	3140	4405	160	7705
CompCert	111k	66k	53k	230k

reusing the OCamlGraph [9] library, with minor adaptations to fit our context: to handle backward analyses, we compute a valid set of entry nodes of the reverse CFG, on which we begin the discovery of the SCC.

Table 1 gives the size of our development, splitting figures according to whether they correspond to Coq definitions, Coq proofs, or OCaml definitions. This consists in around 3.1k lines of code for definitions, 4.4k for the proofs. The whole part related to WTO (specification, computation, validation, and proofs) is about 1.8k lines of Coq and OCaml code. The fixpoint solvers’ specification, implementation and correctness proofs is the largest part of our development: 3.4k lines of Coq for the iterative strategy and 2.5k lines for the recursive one. We also indicate in Table 1 the size of the underlying CompCert code base. This represents 111k lines of Coq definitions and 66k lines of proofs, along with 53k lines of OCaml, for a total of about 230k lines.

7.2 Benchmark Programs

We evaluate the solvers using example C programs from CompCert’s test suite, which includes Spass, a prover for first-order logic, a ray tracer, a compression library and a set of cryptographic primitives. Their sizes range from a few lines of code to several thousands, including programs with many nested loops.

CompCert’s test suite are realistic hand-written C programs. As such, they rarely include very large functions, but are instead made of a large collection of relatively small functions. For Spass, the largest function has 1245 RTL instructions. Only 5 functions have more than 500 instructions, and the average size of a function is 16 instructions. Hence, in order to evaluate and compare the solvers as the size of the analyzed functions grows up, we also include 10 random C programs generated with Csmith [27]. The largest generated function has 3916 RTL instructions, and the average function size is 118 instructions.

7.3 Dataflow Solvers

We conduct experiments on the three following solvers, each of them having two variants for forward and backward analysis. We use the following abbreviations to refer to each of them: K is the Kildall solver as implemented in CompCert, BIter

is our WTO-iterative solver, and BRec is our WTO-recursive solver. The WTO we use in our solvers is always computed using the SCC method (the enumeration method described in Section 3.2 is indeed clearly inefficient in practice). As explained Section 1, the numbering of the CFG nodes makes a difference for the solvers. Indeed, Kildall’s implementation in CompCert is highly sensitive to the numbering of nodes, and converges faster when nodes are numbered according to a reverse postorder (RPO) traversal of the CFG. Our experiments always use an RPO numbering of the CFG, which also leads to better SCC-based WTO.

7.4 Global Compilation Time and Optimization Opportunities

Changing the dataflow algorithm of the compiler can influence globally compilation times. We measure the whole compilation time, for each C source file, and for three distinct compilation chains, with respective solvers K, BIter and BRec. Results are given in Fig. 1. Each point compares compilation time measures for a single file. The (x, y) coordinates correspond to the time measures for each algorithms. Total compilation time with BRec is higher than with K by about 15%. BIter is also slightly more efficient than BRec by about 2%. These compilation times for BRec and BIter both include the WTO validation times. Removing the WTO validation time from compilation time with BRec reduces the overhead against K down to approximately 5%.

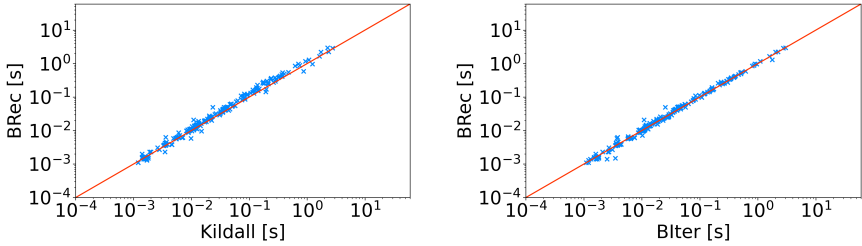


Fig. 1: Total compilation time, by C file (in seconds, log/log scale). Left: K vs. BRec. Right: BIter vs. BRec.

Using a different solver algorithm can in theory influence the quality of compiled programs: more precise analyses results could lead to further optimized programs. In our case, however, for the three solvers, resulting assembly programs are syntactically identical, and their running time would hence be the same. Note that obtaining identical assembly programs does not necessarily imply that dataflow solvers have indeed equal precision. Instead, it only means that program optimizations are currently insensitive to potential variations in the precision of the underlying analyses.

These first, high-level experiments show that BIter and BRec have acceptable performances in practice, both in time and precision. Now, in order to gain a better understanding, we compare solvers with more fine-grained experiments.

7.5 Efficiency of the Solvers

We now study the solvers themselves, independently of program optimizations.

Analysis Time. We assess and compare the analysis time for the solvers, for each analysis, and function by function. We include in the analysis time the fixpoint computation time, and the preprocessing time required to compute the fixpoint: either the WTO computation and validation time for BIter and BRec solvers, or the RPO numbering for K solvers. For long analysis times, BRec performs better, while K is better for short analysis times. But, as compilation time is linearly correlated with the size of the CFG (in number of nodes), this means that, as the size of the function increases, the gain of iterating more efficiently is worth the cost of computing and validating the WTO itself. Solvers BRec and BIter scale better with function size. In conclusion, there is an improvement on the fixpoint computation thanks to more efficient iterations, but this improvement is limited by an overhead due to the computation of the WTO.

Analysis Costs. We consider dataflow solvers computation costs that are relative to transfer functions and join operators, rather than just wall-clock time. We count the number of applications of T and $\sqcup$ when computing fixpoints during the various analyses. Abstracting over the time spent in the particular implementation of T and $\sqcup$ allows us gaining some insights about computation costs inherent to the fixpoint algorithm, independently of a particular analysis.

Our full results appear in our experiment report [16]. In a nutshell, BRec applies more often transfer functions T than K does. However, BRec applies drastically less $\sqcup$ operators than K does. Indeed, when analyzing a node, BRec retrieves dataflow facts from its predecessors, after applying the transfer function to each of them: T needs to be applied once per predecessor each time we analyze a node. Instead, K starts by applying the transfer function of the node, and then propagates the output dataflow fact to all of its successors. In contrast, BRec only applies $\sqcup$ for component heads and for nodes having several predecessors, while K needs to apply it when analysing every node, to join the output dataflow fact with the fact previously computed for that node.

Finally, we compare BIter costs against BRec costs in terms of join operators. Both have similar costs for forward dataflow analyses, but differ for backward analyses. BIter is almost always better than BRec on backward analyses. This is due to the fact that SCC are discovered from a set of exit nodes instead of an entry node, as explained in Section 4. As a result, WTO component heads often correspond to the ends of loop bodies, rather than being at loop headers. In such a situation, BRec misses information from yet unprocessed nodes more often than BIter does. Indeed BRec might do a large number of iterations on a subcomponent while missing information coming from nodes located after it.

In contrast, BIter performs iterations on the whole outermost component, thus converging faster. This phenomenon is amplified for analyses like Allocation, which consider all nodes of the CFG as exit points.

7.6 Precision of the Solvers

We evaluate the precision of the solvers by comparing the abstract values they return, for each analysis, and for each function. Comparison is done using the partial order from the analysis pseudo-semi-lattice domain. Differences can occur because CompCert uses pseudo-semi-lattices instead of semi-lattices: we have no guarantee that $\sqcup$ returns the *least* upper bound, and over-approximations might happen depending on the processing order of the nodes. We observe a few differences in terms of precision. For domains where $\sqcup$ effectively computes the least upper bound, such as for Liveness or Allocation, we observe no difference across the solvers. However, the three solvers do not always compute the same result for Value Analysis where the pseudo-semi-lattice is not a semi-lattice. On Value Analysis, for 6.1% of the 5766 analyzed functions, one of the three algorithms outputs at least one different abstract value from the two others at some node of the CFG.

8 Related Work

This work is at the intersection of two research areas. The first one focuses on dataflow analysis algorithms, where the algorithms are usually designed for specific cases: for instance, the abstract structures almost never have infinite ascending chains. It is the case in CompCert: all its optimizations rely on abstract domains with length-bounded ascending chains. The second area concerns generic fixpoint solvers for static analysis with abstract interpretation, where Bourdoncle’s method is often the reference approach. Indeed it allows to do *widening* to speed up the convergence of the analysis and to guarantee its termination. It is a need since those solvers potentially have to deal with lattices with very long or even infinite ascending chains. Such solvers are usually described in a theoretical context, and are not especially designed for dataflow analysis.

8.1 Dataflow Analysis

The main and historical approach to dataflow analysis is based on worklists [13,21]. A formalization and proof of correctness in Coq is available for Kildall’s method in the existing dataflow analysis framework of CompCert [20].

To simplify dataflow analysis on *reducible graphs*, one can decompose them in *intervals* of nodes [2,3], that describe the main blocks of the underlying program. It then suffices to compute the abstract information at the head of an interval Bourdoncle [5] describes a method to compute a WTO from an interval decomposition of a reducible CFG. However, optimization passes often make the CFG

irreducible. It is thus difficult to use such an approach with optimized programs. As a result, in our work, we did not choose to implement this method.

The experimental evaluation of Cooper et al. [10] for worklist algorithms addresses similar questions to our evaluation of Bourdoncle’s method. Namely, they both assess the impact of the order in which a dataflow solver process the nodes on the efficiency of the solver. Their main conclusion is that the efficiency of these solvers strongly depends on the data structure chosen for the worklist. We can draw a parallel with the fact that the efficiency of Bourdoncle’s method strongly depends on how we compute the WTO. In both cases, the efficiency of dataflow analysis algorithms mostly hinges on the choice of the iteration order.

Bourdoncle’s method, and more generally abstract interpretation techniques, bring an interesting approach to dataflow analysis as alternatives to historical algorithms. This new approach have already been partially explored by Sherman [24] to improve the Soot Java optimization framework [26]. Indeed Soot was initially based on Kildall’s algorithm and did not support widening. As a result, it was difficult to adapt it to analyses where the lattices have long ascending chains. Thus, Soot was not commonly used for program verification, or for complex program optimizations such as array bound check elimination [22]. In her work, Sherman redesign the dataflow analysis interface of Soot. The new interface allows the implementation of abstract interpretation-based methods such as Bourdoncle’s one. Note that her work only focuses on forward dataflow analyses.

8.2 Generic Solvers for Static Analysis

Bourdoncle’s iterations have many applications in the field of static analysis. In a recent work, Lemerre [18] used the recursive strategy of Bourdoncle to compute a SSA abstraction of a program. His algorithm is formalized on a theoretical toy language and implemented as a plugin for FRAMA-C which translates C to a SSA intermediate representation of LLVM. Blazy et al. [4] developed a formally verified value analysis on an intermediate representation of CompCert. This analysis is based on abstract interpretation. They choose to implement their fixpoint solver with Bourdoncle’s method. Indeed, they perform an analysis with the lattice of bounded intervals. Since this lattice contains infinite ascending chains, the analysis required widening. Thus they could not reuse the Kildall framework from CompCert. However their fixpoint solver is validated *a posteriori*. Our paper presents a partial proof of correctness in Coq of the fixpoint computation (termination is not proved but guaranteed by the introduction of *fuel*).

Local solvers are often used to do static program analysis. They consist in computing information at some node only by exploring its dependencies. Thus, the problem they address is different from our problem: the interface we consider here requires an analysis result on all the nodes of the graph. Some program optimizations techniques also use local solvers to perform, e.g. liveness queries [17], but this is not the case for CompCert’s general dataflow framework.

An historical example of local solver is the top-down solver (**TD**) [7]. The idea is to simulate chaotic iterations from a node while building a graph of the

dependencies of this node. The simulations are done recursively: when a dependency p of a node n is encountered, a sub-simulation is performed on p . The abstract values computed are updated each time a sub-simulation is done. The sub-simulation on p has to be re-done if the abstract value at n changed in the meantime. The algorithm **RLD** (for Recursive Local Dependencies) [23,11] reuses the same idea but handles slightly more precisely the dependencies. Indeed, when the value at a node n is changed, **TD** removes recursively all the nodes that depends on n from the list of solved nodes and has to solve them again. **RLD** only removes the successors of n from the list of solved nodes. **RLD** has been formalized in Coq by Hofmann [11].

These algorithms indeed follow a “top-down” approach: they start from the node to analyze (the top) and go down as they analyze recursively the dependencies. “Bottom-up” algorithms rather consist in computing a fixpoint on a pre-determined set of nodes containing the node to analyze. Worklist algorithms are examples of “bottom-up” algorithms.

Kim et al. recently presented a variant of Bourdoncle’s method to parallelize the fixpoint computation [14]. This variant is based on the notion of *weak partial order*, a generalization of the WTO of Bourdoncle. While a WTO is a total order on the reachable nodes of the graph, a WPO is a partial order. Thus, non-comparable nodes can be processed in parallel since they are not interdependent.

9 Conclusion and Perspectives

This work provides a formalization of Bourdoncle’s WTO-based method to solve dataflow problems. This work is supported by an implementation which is verified within the Coq proof assistant, and fits into a realistic context: compiler optimizations in CompCert. We achieved performances that are comparable to those of Kildall’s historical algorithm for dataflow analysis, while keeping the same level of precision. Finally, our formal proof allows for a better understanding of the correctness arguments of the solvers.

Our dataflow solver could be improved in several directions. First, we could implement and verify the SCC algorithm directly in Coq to compute the WTO, thus avoiding the non-negligible overhead of the validator. We could adapt the formally verified implementation of Tarjan’s SCC algorithm by Chen et al. [8] to fit the context of WTO computation. Second, we could refine our implementation by distinguishing upper-bound and widening operators. Depending on whether a node is a widening point, identified by component heads in the WTO, applying a widening operator rather than a simple upper bound operator could make the analyses converge faster in practice, with no consequence on our formal correctness proof if we keep a fuel-bounded recursion technique.

Another avenue for research, from a formal proof perspective, would be to specify widening operators in a more precise way to study the termination and other related convergence notions of the solver algorithms.

Data Availability Statement

The software artifact containing our Coq development, its integration into CompCert, and our framework for experimental evaluation, is available online [15].

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (Jul 2006)
2. Allen, F.E., Cocke, J.: A program data flow analysis procedure. *Communications of the ACM* **19**(3), 137 (Mar 1976). <https://doi.org/10.1145/360018.360025>
3. Allen, F.E.: Control flow analysis. *SIGPLAN Not.* **5**(7), 1–19 (jul 1970). <https://doi.org/10.1145/390013.808479>
4. Blazy, S., Laporte, V., Maroneze, A., Pichardie, D.: Formal verification of a C value analysis based on abstract interpretation. In: Fahndrich, M., Logozzo, F. (eds.) *SAS - 20th Static Analysis Symposium*. 7935, vol. *Lecture Notes in Computer Science*, pp. 324–344. Springer, Seattle, United States (Jun 2013). https://doi.org/10.1007/978-3-642-38856-9_18
5. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) *Formal Methods in Programming and Their Applications*. pp. 128–141. Springer Berlin Heidelberg, Berlin, Heidelberg (1993). <https://doi.org/10.1007/BFb0039704>
6. Bourdoncle, F.J.: *Sémantiques Des Langages Impératifs d'ordre Supérieur et Interprétation Abstraite*. Ph.D. thesis, Palaiseau, Ecole polytechnique (1992)
7. Charlier, B.L., Van Hentenryck, P.: A universal top-down fixpoint algorithm. Tech. rep., Brown University, USA (1992)
8. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of tarjan's strongly connected components algorithm in why3, coq and isabelle. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA. LIPICs*, vol. 141, pp. 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPICS.ITP.2019.13>
9. Conchon, S., Filliâtre, J.C., Signoles, J.: Designing a generic graph library using ML functors (04 2007)
10. Cooper, K.D., Harvey, T.J., Kennedy, K.: An empirical study of iterative data-flow analysis. In: *2006 15th International Conference on Computing*. pp. 266–276 (2006). <https://doi.org/10.1109/CIC.2006.22>
11. Hofmann, M., Karbyshev, A., Seidl, H.: Verifying a local generic solver in Coq. In: t, R., Martel, M. (eds.) *Static Analysis*. pp. 340–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_21
12. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified c static analyzer. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. p. 247–259. POPL '15, Association for Computing Machinery, New York, NY, USA (2015). <https://doi.org/10.1145/2676726.2676966>
13. Kildall, G.A.: A unified approach to global program optimization. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. p. 194–206. POPL '73, Association for Computing Machinery, New York, NY, USA (1973). <https://doi.org/10.1145/512927.512945>

14. Kim, S.K., Venet, A.J., Thakur, A.V.: Deterministic parallel fixpoint computation. *Proceedings of the ACM on Programming Languages* **4**(POPL), 1–33 (Jan 2020). <https://doi.org/10.1145/3371082>
15. La Spina, R., Demange, D., Blazy, S.: Formal Verification of WTO-based Dataflow Solvers - Artifact (Feb 2025). <https://doi.org/10.5281/zenodo.14587599>
16. La Spina, R., Demange, D., Blazy, S.: Formal Verification of WTO-based Dataflow Solvers: Artifact Experience Report. In: *Proceedings of European Symposium on Programming (ESOP)*. LNCS(2025). https://doi.org/10.1007/978-3-031-91121-7_5
17. Léchenet, J.C., Blazy, S., Pichardie, D.: A Fast Verified Liveness Analysis in SSA Form. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *Automated Reasoning*, vol. 12167, pp. 324–340. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_19
18. Lemerre, M.: SSA Translation Is an Abstract Interpretation. *Proceedings of the ACM on Programming Languages* **7**(POPL), 1895–1924 (Jan 2023). <https://doi.org/10.1145/3571258>
19. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115 (jul 2009). <https://doi.org/10.1145/1538788.1538814>
20. Leroy, X.: A formally verified compiler back-end. *Journal of Automated Reasoning* **43**(4), 363–446 (2009). <https://doi.org/10.1007/s10817-009-9155-4>
21. Nielson, F., Nielson, H., Hankin, C.: *Principles of Program Analysis*. Springer Publishing Company, Incorporated (Jan 1999). <https://doi.org/10.1007/978-3-662-03811-6>
22. Qian, F., Hendren, L., Verbrugge, C.: A Comprehensive Approach to Array Bounds Check Elimination for Java. In: Goos, G., Hartmanis, J., Van Leeuwen, J., Horspool, R.N. (eds.) *Compiler Construction*, vol. 2304, pp. 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_23
23. Seidl, H., Wilhelm, R., Hack, S.: *Compiler Design: Analysis and Transformation*. Springer Publishing Company, Incorporated (11 2013). <https://doi.org/10.1007/978-3-642-17548-0>
24. Sherman, E.: Redesigning Soot’s data-flow analysis framework for abstract interpretation. In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. pp. 78–84. ACM, Amsterdam Netherlands (Jul 2018). <https://doi.org/10.1145/3236454.3236506>
25. Tarjan, R.: Depth-first search and linear graph algorithms. *SIAM Journal on Computing* **1**(2), 146–160 (1972). <https://doi.org/10.1137/0201010>
26. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A Java bytecode optimization framework. In: *CASCON First Decade High Impact Papers on - CASCON ’10*. pp. 214–224. ACM Press, Toronto, Ontario, Canada (2010). <https://doi.org/10.1145/1925805.1925818>
27. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. *SIGPLAN Not.* **46**(6), 283–294 (jun 2011). <https://doi.org/10.1145/1993316.1993532>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Formal Verification of WTO-based Dataflow Solvers Artifact Experience Report

Roméo La Spina^{}, Delphine Demange^{}, and Sandrine Blazy^{} ^{}

Univ Rennes, Inria, CNRS, IRISA, Rennes, France
`{romeo.la-spina,delphine.demange,sandrine.blazy}@irisa.fr`

Abstract. In this report, we provide our full set of results for the experimental evaluation of our formally verified, WTO-based dataflow solvers [2]. We also detail useful information regarding our artifact [3] and evaluation setup. In particular, we validate experimentally that our implementation of Bourdoncle’s iteration strategies have acceptable performances in practice, when applied to the dataflow analyses used in the CompCert C verified compiler. Our experiments also help better understanding the differences between the solvers, their strengths and their weaknesses.

1 Experimental Data

Our baseline for comparison is the generic dataflow solver from the CompCert C compiler [4] version 3.13, which implements Kildall’s algorithm [1]. Below we describe the dataflow analyses implemented in CompCert, as well as the set of program benchmarks we used to evaluate our solvers.

1.1 Dataflow Analyses used by Compiler Passes

CompCert relies on several dataflow analyses operating over different intermediate representations. All of these represent programs as control-flow graphs (CFG). Common optimizations, *e.g.*, constant propagation, or common subexpression elimination (CSE), are performed on the RTL representation. Register allocation is performed on the XTL representation, and the linearization of the CFG is performed on the LTL representation. Depending on the analysis problem, the CFG must be traversed forwards or backwards.

Table 1 sums up representative dataflow analyses from CompCert, their abstract domains and the compilation passes to which they are related. Forward analyses always start the CFG traversal from the CFG entry node. Backward NS refers to backward analyses starting from an over-approximated node set (NS) for exit nodes. Backward AN refers to backward analyses where all nodes (AN) are considered as exit nodes.

Table 1: Dataflow analyses in CompCert (NS = Node Set, AN = All Nodes).

Analysis name	Direction	Abstract domain (pseudo-semi-lattice L)	Compilation passes
Value Analysis	Forward	Mappings from regs./mem. locations to abstract values	Constant propagation Redundancy elimination CSE
Liveness	Backward NS	Sets of RTL registers	Constant propagation Redundancy elimination CSE
Neededness	Backward NS	Mappings from regs./mem. locations to abstract neededness information	Redundancy elimination
Splitting	Backward NS	Mappings from registers to live ranges	Register allocation
Regalloc Liveness	Backward NS	Sets of XTL registers	Register allocation
Allocation	Backward AN	Sets of equations on registers/memory locations	Register allocation
Linearize	Forward	Boolean	CFG linearization

Table 2: Tests suites. A functions size counts the instructions in the RTL CFG.

Test suite	Number of functions	Average function size	Largest function
Spass	4309	16	1245
Ray	108	36	733
Comp	60	40	139
Misc	122	56	448
Reg	889	71	1164
Random	291	118	3916
Total	5766	32	3916

1.2 Test Suites

We evaluate our solvers against six test suites, which we list in Table 2. The first five suits are taken from CompCert’s distribution: Spass is the prover for first-order logic Spass [5], Ray is a ray tracer, Comp is a compression library, Misc is a set of cryptographic primitives (AES, SHA1, SHA3) and miscellaneous C programs (incl. FFT), and Reg corresponds to regression testing programs.

We added a sixth suite, Random, made of 10 random C programs generated with Csmith [6]. This suite allows us evaluating the scalability of our solvers as the size of the analyzed functions grows up. The biggest function in Random has 3916 RTL instructions while functions in the other suites are up to 1245 RTL instructions.

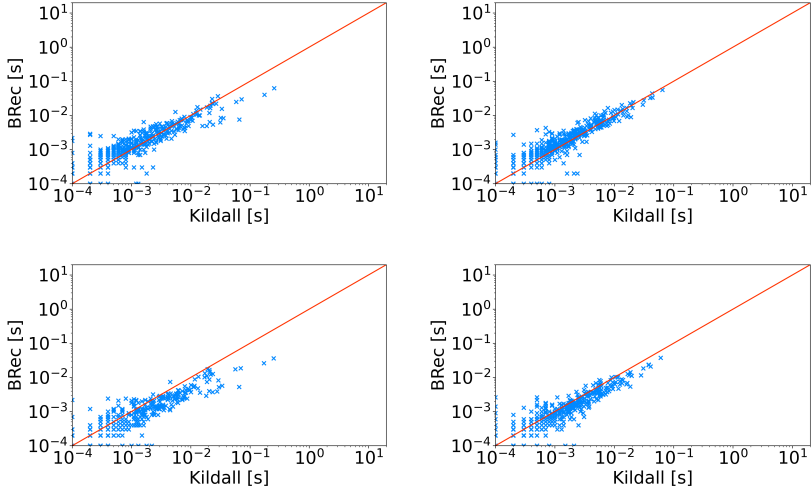


Fig. 1: Analysis time, by function (in seconds, log/log scale). Comparing K with BRec. Left: ValueAnalysis (forward). Right: Liveness (backward). Top: preprocessing time included. Bottom: fixpoint computation time only.

2 Experimental Results

We use the following abbreviations to refer to each of the solver we consider in this report: K is the Kildall solver as implemented in CompCert, BIter is our WTO-iterative solver, and BRec is our WTO-recursive solver.

Total compilation times are discussed in the paper [2]; they show that BIter and BRec have acceptable performance in practice. Below, we provide detailed results for the finer-grained comparison of our solvers.

2.1 Efficiency of the Solvers: Analysis Times

We compare the analysis time of the solvers, for each analysis, and function by function. Our detailed results are given in Fig. 1 for ValueAnalysis and Liveness, which are good representatives of forward and backward analyses, respectively. In Fig. 1, the points compare analysis time measures for a single function. The (x, y) coordinates correspond to the time measures for each algorithm.

For long analysis times, BRec performs better, while K is better for short analysis times. But, as compilation time is linearly correlated with the size of the CFG (in number of nodes, *i.e.*, instructions), this means that, as the size of the function increases, the gain of iterating more efficiently is worth the cost of computing and validating the WTO itself. Solvers BRec and BIter scale better with function size. Our interpretation is confirmed by the difference between plots which include preprocessing time (top ones) and the ones that only include

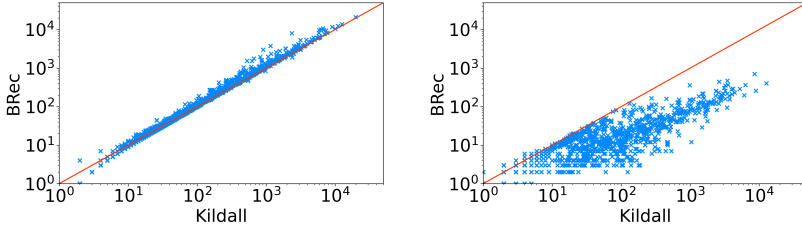


Fig. 2: Analysis costs, by function, for ValueAnalysis (log/log scale). BRec versus K. Left: number of application of transfer functions T . Right: number of application of join operator $\sqcup$.

the fixpoint computation time (bottom ones). Indeed, if we only compare fixpoint computation times, BRec scales better than K (see the bottom left plot of Fig. 1).

In conclusion, there is an improvement on the fixpoint computation thanks to more efficient iterations, but this improvement is limited by an overhead due to the computation of the WTO.

2.2 Efficiency of the Solvers: Analysis Costs

In addition to wall-clock analysis times, we also consider dataflow solvers computation costs that are relative to the number of transfer functions (T) application and join operators ($\sqcup$). Figure 2 shows the dataflow solver costs in terms of transfer functions and join operators, comparing K with BRec. We only show results for ValueAnalysis, they are similar for other analyses, either forward or backward.

BRec applies more often transfer functions T than K does. However, BRec applies drastically less $\sqcup$ operators. Indeed, when analyzing a node, BRec retrieves dataflow facts from its predecessors, after applying the transfer function to each of them: T needs to be applied once per predecessor each time we analyze a node. Instead, K starts by applying the transfer function of the node, and then propagates the output dataflow fact to all of its successors. Moreover, BRec only applies $\sqcup$ for component heads and for nodes having several predecessors, while K needs to apply it when analyzing every node, to join the output dataflow fact with the fact previously computed for that node.

2.3 Precision of the Solvers

We evaluate the precision of the solvers by comparing the abstract values they return, for each analysis, and for each function. Comparison is done using the partial order from the analysis pseudo-semi-lattice domain.

The three solvers return almost always the same value, except for a few functions (6.1% of 5766 analyzed functions) on domains where the join operator

Table 3: Comparing the precision of dataflow algorithms for Value Analysis. For instance, there are 34 functions where BRec is more precise than K (at more than one CFG node), and this represents 5.9‰ of the analyzed functions.

more precise than	K	BIter	BRec
K	—	4 (0.7‰)	0 (0‰)
BIter	31 (5.3‰)	—	23 (3.9‰)
BRec	34 (5.9‰)	4 (0.7‰)	—

over-approximates the least upper bound. More precisely, Table 3 compares the precisions of the dataflow solvers, two-by-two, for Value Analysis. For each pair of solvers (A, B), we give the number of functions where A is more precise than B on at least one node of the CFG. Indeed, differences tend to propagate to large parts of the CFG, so we do not consider how many nodes in the CFG give rise to different abstract values.

There are very few differences, and they tend to be in favor of Bourdoncle’s method BIter and BRec. These results are to be put in perspective: the differences affect a small number of functions. Additionally, to date, CompCert’s optimizations do not take advantage of these precision gains.

Data Availability Statement

The software artifact containing our Coq development, its integration into CompCert, and our framework for experimental evaluation, is available online [3].

References

1. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 194–206. POPL ’73, Association for Computing Machinery, New York, NY, USA (1973). <https://doi.org/10.1145/512927.512945>
2. La Spina, R., Demange, D., Blazy, S.: Formal verification of WTO-based dataflow solvers. In: Proc. of European Symposium on Programming (ESOP). LNCS (2025). https://doi.org/10.1007/978-3-031-91121-7_4
3. La Spina, R., Demange, D., Blazy, S.: Formal Verification of WTO-based Dataflow Solvers - Artifact (Feb 2025). <https://doi.org/10.5281/zenodo.14587599>
4. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (jul 2009). <https://doi.org/10.1145/1538788.1538814>
5. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. pp. 140–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
6. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. SIGPLAN Not. **46**(6), 283–294 (jun 2011). <https://doi.org/10.1145/1993316.1993532>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Multiparty Session Types with a Bang!

Matthew Alan Le Brun^(✉), Simon Fowler, and Ornela Dardha

University of Glasgow, Glasgow, UK

m.le-brun.1@research.gla.ac.uk

{simon.fowler,ornela.dardha}@glasgow.ac.uk

Abstract. Replication is an alternative construct to recursion for describing infinite behaviours in the π -calculus. In this paper we explore the implications of including type-level replication in *Multiparty Session Types* (MPST), a behavioural type theory for message-passing programs. We introduce MPST!, a session-typed multiparty process calculus with replication and first-class roles. We show that replication is *not* an equivalent alternative to recursion in MPST, and that using both replication *and* recursion in one type system in fact allows us to express both context-free protocols and protocols that support mutual exclusion and races. We demonstrate the expressiveness of MPST! on examples including binary tree serialisation, dining philosophers, and a model of an auction, and explore the implications of replication on the decidability of typechecking.

Keywords: Multiparty session types, Replication, Distributed protocols

1 Introduction

Our world is powered by a multitude of computer systems working together by *communicating*, *i.e.*, sending and receiving messages according to some *protocol*. It is therefore vital to *verify* the correctness of both communication protocols and their implementations, to ensure our programs behave according to their specifications, and to guarantee that specifications are indeed *safe*.

Session types [9,14,15,31] provide a lightweight method by which a developer can ensure safety of, and conformance to, communication protocols. Session types can be thought of as *types for protocols* which can be attached to a communication *channel* to specify *how* it should be used, and can be used to detect issues such as *communication mismatches* and *deadlocks* early in the development process. *Multiparty session types* (MPST) [7,16,25] generalise binary session types to allow reasoning about communication between two *or more* participants, and have been shown to be expressive enough to capture a range of practical protocols such as the OAuth 2 authentication protocol [27].

Example 1 (Client-Server-Worker). Using generalised MPST [27], we describe the types for three participants in a simple work-offloading system.

$$\begin{aligned} S_c &:= \textcolor{blue}{s} \oplus \text{req}(\text{int}) . \textcolor{blue}{w} \&\text{ans}(\text{str}) \\ S_s &:= \textcolor{blue}{c} \&\text{req}(\text{int}) . \textcolor{blue}{w} \oplus \text{fw}(\text{int}) & S_w &:= \textcolor{blue}{s} \&\text{fw}(\text{int}) . \textcolor{blue}{c} \oplus \text{ans}(\text{str}) \end{aligned} \tag{1}$$

The above describes types for a *client*, *server*, and *worker* respectively. The *client*, having type S_c , sends ($\oplus$) a *request* to the *server* with payload type *int*, then ($\cdot$) waits to receive ($\&$) an *answer* from the *worker* with payload type *str*. The *server*, upon receiving the *request* from the *client*, *forwards* it to the *worker*. Lastly the *worker*, after receiving the *forwarded* request from the *server*, sends the *answer* to the *client*. A MPST system verifies that any written program code conforms to this specification (known as *session fidelity*), and that this protocol is *safe*—*i.e.*, that processes send and receive messages of compatible types.

Despite the potential of MPST for safe distributed programming, there remain limitations to the theory that impede their adoption for practical systems. For instance, generalising Example 1 to multiple workers in the style of a load-balancer is non-trivial and has inspired a series of work on the generalisation of *direction of choice* [28]. Further, generalising the number of *clients* is also non-trivial—typically, MPST theories assume a global view of *all* participants in a session. Lastly, the *objects* of actions in MPST (e.g., the recipient of a sent message) are *hard coded* because role names are *constants*.

Example 2 (Load Balancer for n clients). We introduce *replication* and *first-class roles*, combined with undirected choice in sends, to generalise Example 1 to support *two workers* and *any number* of clients.

$$\begin{aligned} S_s &:= !\alpha \& \text{req}(\text{int}) \cdot \oplus \left\{ \begin{array}{l} w_1 \text{fw}(\text{int}, \alpha) \cdot \alpha \oplus \text{wrk}(w_1) \\ w_2 \text{fw}(\text{int}, \alpha) \cdot \alpha \oplus \text{wrk}(w_2) \end{array} \right. \\ S_{w_i} &:= !s \& \text{fw}(\text{int}, \gamma) \cdot \gamma \oplus \text{ans}(\text{str}) \quad \text{for } i \in \{1, 2\} \end{aligned} \quad (2)$$

The first difference is the use of the *bang* (!) operator to denote a *replicated* action, *i.e.*, one which may occur *any number* of times. This makes the server agnostic to the *number* of requests. Second, a server now waits for requests—not from a specific client—but from *any* participant, binding the name of the sender to role variable α . This makes the server agnostic to the *source* of the requests. The server then makes a *choice* to forward the request to one of two workers, notably whilst passing the name of the client as one of the payloads. Finally, the server informs the client of the choice it made by sending the name of the worker in that branch. The worker type is also updated to be replicated, as it is dependent on the number of requests forwarded by the server. Notably, it receives the name of the client in the forward message, binding it to γ , and uses it to send the final answer. A client may now be defined as:

$$S_{c_j} := s \oplus \text{req}(\text{int}) \cdot s \& \text{wrk}(\omega) \cdot \omega \& \text{ans}(\text{str}) \quad \text{for any } j \in \mathbb{N} \quad (3)$$

As a result of the *replicated types* and *first-class role names* on the server-side, we may instantiate *any number of clients* and have them make *any number of requests*—all without changing the server-side protocol. Conversely, updating the number of workers has *no impact* on the types of clients. Thus, this extension promotes *modular* design of components in multiparty systems.

In fact, as we will see in Section 3, the addition of replication—especially when it is used in tandem with recursion—has several surprising consequences, in particular allowing us to describe *context-free protocols* as well as protocols that deal with *races* and *mutual exclusion*.

$c ::= s[q] \mid x$	(session w / role, channel variable)
$\rho ::= q \mid \alpha$	(role name value, role name variable)
$a ::= c \mid \alpha \quad V ::= c \mid \rho$	(names, values)
$b ::= x \mid \alpha \quad d ::= s[q] \mid q$	(binders, concrete values)
$P, Q ::= P \mid Q \mid (\nu s) P \mid \mathbf{0}$	(composition, restriction, termination)
$\mid \sum_{i \in I} c_i[\rho_i] \oplus m_i(\tilde{V}_i) . P_i$	(choice of sends)
$\mid [!]\ c[\rho] \&_{i \in I} m_i(\tilde{b}_i) . P_i$	([replicated] branching receive)
$\mid \text{def } D \text{ in } Q \mid X\langle \tilde{c} \rangle$	(process definition, process call)
$D ::= X(\tilde{x}) = P$	(process declaration)

Fig. 1. Syntax of MPST!

Contributions. The overarching contribution of this paper is the first integration of replication and first-class roles into a generalised MPST calculus, and an exploration of the impacts of these extensions on expressiveness and decidability. Our specific contributions are as follows:

1. We present MPST!, the first multiparty session-typed language with *replication* and *first-class roles* (Section 2), and prove its *metatheory* in the form of *subject reduction* and *session fidelity* properties (Section 2.4).
2. We show several expressiveness results through a series of representative examples (Section 3.1): in particular, replication lifts the expressive power of types and thus we give the first account of *context-free* MPST. We show that combining both replication and recursion allows us to model races and mutual exclusion; we demonstrate nontrivial examples including *binary tree serialisation*, the *dining philosophers problem*, and an *auction service*.
3. We demonstrate the impacts of replication on the decidability of typechecking (Section 3.2). We show that the decidability of typechecking is contingent on the decidability of a given safety property, and demonstrate conditions guaranteeing a property to be decidable. Finally we show two syntactic approximations to allow us to verify that a property is decidable.

Section 4 gives an account of related work and Section 5 concludes. Detailed proofs of all our presented theorems can be found in the technical report [20].

2 Multiparty Session Types with a Bang!

In this section we introduce MPST!, a conservative extension of existing multiparty session calculi [7,25,27] with support for *replication* and *first-class roles*.

2.1 Language

Figure 1 shows the syntax of MPST!.

Names, values, and binders. A *session name*, ranged over by $s, s', \dots$, represents a collection of interconnected participants. A *role* is a participant in a multiparty communication protocol, and each *communication endpoint* $s[q]$ is obtained by indexing a session name with a role. In contrast to existing MPST calculi, MPST! supports *first-class* roles, meaning that a role may be communicated as part of a message. To this end, a role ρ may either be a concrete role p (e.g., s or w in our load balancing example) or a *role variable* α . A name a is either an endpoint, a variable, or a role variable, whereas a value V is either a channel or a role. Binders b are used when receiving a message and can either be a variable binder or a role variable binder. Concrete values d are used when sending a message (at runtime) and are either an endpoint or a role name value.

Processes. Processes are ranged over by $P, Q, R, \dots$: process $P|Q$ denotes P and Q running in parallel; *session restriction* $(\nu s)P$ binds session name s in process P ; and $\mathbf{0}$ is the inactive process.

As in the π -calculus, but unlike other MPST calculi, MPST! supports *output-guarded choice* $\sum_{i \in I} c_i[\rho_i] \oplus m_i(\tilde{V}_i) \cdot P_i$, allowing a nondeterministic send along any c_i to role ρ_i with label m_i and payload $\tilde{V}_i$, with the process continuing as P_i . *Branching receive* $c[\rho] \&_{i \in I} m_i(b_i) \cdot P_i$ denotes a process waiting on channel c for one of a set of messages from role ρ with label m_i , binding the received data to b_i before continuing according to P_i . It is key to note that the object of a communication action is indicated via ρ (for both sending and receiving), which can either be a concrete value, or a role variable. The second key difference is the optional use of the *bang* ! with a branching receive, marking it as *replicated*—i.e., it may be used 0 or more times, modelling *infinitely available servers*.

Definition 1 (Reduction context). A reduction context $\mathbb{C}$ is given as: $\mathbb{C} ::= \mathbb{C}|P \mid (\nu s)\mathbb{C} \mid \text{def } D \text{ in } \mathbb{C} \mid []$.

A *reduction context* allows us to evaluate processes under parallel composition and name restrictions. With this, reduction rules on processes are given in Figure 2. The rules make use of a standard *structural congruence* $\equiv$ [20, Appendix A] that allows us to treat parallel composition as commutative and associative, as well as including the usual π -calculus scope extrusion rule.

Rule [R-C] shows synchronous communication between two processes in session s . The first process, playing role p , offers role q a choice of message labels and associated process continuations. The second process, playing role q , sends a message with label m_k and transmits payloads $\tilde{d}$. The first process reduces to the selected continuation with the transmitted payloads substituted for the binders in the selected branch, and the second process reduces to the continuation Q .

Rules [R-!C₁] and [R-!C₂] describe communication with a *replicated* process R . Rule [R-!C₁] is similar to R-C but the replicated process remains unchanged and the continuation Q_k is evaluated in parallel. Rule [R-!C₂] handles the case where the replicated process does not need to receive from a specific role, but instead allows communication with an *arbitrary* role: the rule binds the sending role to α in the replicated continuation. We refer to this as a *universal receive*.

Process reduction

$$\boxed{P_1 \rightarrow P_2}$$

$$\text{R-C} \quad s[p][q] \&_{i \in I} m_i(\tilde{b}_i) \cdot P_i \mid s[q][p] \oplus m_k(\tilde{d}) \cdot Q \rightarrow P_k\{\tilde{d}/\tilde{b}_k\} \mid Q \text{ if } k \in I$$

$$\text{R-!C}_1 \quad \frac{R = !s[p][q] \&_{i \in I} m_i(\tilde{b}_i) \cdot Q_i}{s[q][p] \oplus m_k(\tilde{d}) \cdot P \mid R \rightarrow P \mid R \mid Q_k\{\tilde{d}/\tilde{b}_k\}} \text{ if } k \in I$$

$$\text{R-!C}_2 \quad \frac{R = !s[p][\alpha] \&_{i \in I} m_i(\tilde{b}_i) \cdot Q_i}{s[q][p] \oplus m_k(\tilde{d}) \cdot P \mid R \rightarrow P \mid R \mid Q_k\{\tilde{d}/\tilde{b}_k\}\{q/\alpha\}} \text{ if } k \in I$$

$$\text{R-+} \quad \sum_{i \in I} s_i[q_i][p_i] \oplus m_i(\tilde{d}_i) \cdot P_i \rightarrow s_j[q_j][p_j] \oplus m_j(\tilde{d}_j) \cdot P_j \text{ for } j \in I$$

$$\text{R-X} \quad \text{def } X(\tilde{x}) = P \text{ in } (X\langle \widetilde{s'[\mathbf{r}]} \rangle \mid Q) \rightarrow \text{def } X(\tilde{x}) = P \text{ in } (P\langle \widetilde{s'[\mathbf{r}]} \rangle / \tilde{x} \mid Q)$$

$$\begin{array}{c} \text{R-}\equiv \\ \frac{P \equiv P' \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'} \end{array} \qquad \begin{array}{c} \text{R-C} \\ \frac{P \rightarrow P'}{\mathbb{C}[P] \rightarrow \mathbb{C}[P']} \end{array}$$

Fig. 2. Operational semantics for the extended multiparty session π -calculus.

The **[R-+]** rule evaluates a branching output by nondeterministically evaluating to one of the sending branches; rule **[R-X]** handles a recursive call. Finally, rules **[R- $\equiv$]** and **[R-C]** are administrative, allowing reduction modulo structural congruence and under contexts respectively.

Example 3 (Load Balancer: Process Reduction). We recall the load balancer example from Section 1, but this time, we present processes for each role (Figure 3) to demonstrate how our operational semantics handles communication. Consider a single client P_c in parallel with three server-side processes:

$$\begin{array}{c|c|c|c} P_c & P_s & P_{w_1} & P_{w_2} \\ \hline = s[c][s] \oplus \text{req}\langle 42 \rangle \cdot P'_c & !s[s][\alpha] \& \text{req}(x) \cdot P'_s & P_{w_1} & P_{w_2} \end{array}$$

Using **[R-!C₂]**, the client and server reduce. The reduction advances the client to its continuation P'_c , and pulls out a copy of the server's continuation as a new process. It is key to note that α is acting as a binder in P_s , therefore, in the continuation we observe a role variable substitution:

$$\begin{aligned} &\rightarrow P'_c \mid P_s \mid \left(\sum_{i=1}^2 s[s][w_i] \oplus \text{fw}\langle x, \alpha \rangle \cdot P_{s_i}'' \right) \{42/x\}\{c/\alpha\} \mid P_{w_1} \mid P_{w_2} \\ &= P'_c \mid P_s \mid \sum_{i=1}^2 s[s][w_i] \oplus \text{fw}\langle 42, c \rangle \cdot (P_{s_i}'' \{42/x\}\{c/\alpha\}) \mid P_{w_1} \mid P_{w_2} \end{aligned}$$

$$\begin{aligned}
P_c &:= s[c][s] \oplus \text{req}\langle 42 \rangle . s[c][s] \& \text{wrk}(\omega) . s[c][\omega] \& \text{ans}(z) . 0 \\
P_s &:= !s[s][\alpha] \& \text{req}(x) . \sum_{i=1}^2 s[s][w_i] \oplus \text{fw}\langle x, \alpha \rangle . s[s][\alpha] \oplus \text{wrk}\langle w_i \rangle . 0 \\
P_{w_i} &:= !s[w_i][s] \& \text{fw}(y, \gamma) . s[w_i][\gamma] \oplus \text{ans}\langle \text{"life"} \rangle . 0
\end{aligned}$$

Fig. 3. Process definitions for load balancer example

The spawned server process will then non-deterministically choose a worker to send to via rule $[R-+]$; suppose w_1 is picked.

$$\rightarrow P_c' \quad | \quad P_s \quad | \quad s[s][w_1] \oplus \text{fw}\langle 42, c \rangle . P_{s_1}'' \quad | \quad !s[w_1][s] \& \text{fw}(y, \gamma) . P_{w_1}' \quad | \quad P_{w_2}$$

Communication is now possible between the spawned server process and worker w_1 , using rule $[R-!C_1]$. As before, this communication advances the sender process and pulls out a copy of the worker's continuation:

$$\rightarrow P_c' \quad | \quad P_s \quad | \quad P_{s_1}'' \quad | \quad P_{w_1} \quad | \quad P_{w_1}' \{42/y\}\{c/\gamma\} \quad | \quad P_{w_2}$$

The client can now learn which worker was chosen, terminating the spawned server process, then the answer is exchanged between the worker and client:

$$\begin{aligned}
&\rightarrow s[c][w_1] \& \text{ans}(z) . 0 \quad | \quad P_s \quad | \quad 0 \quad | \quad P_{w_1} \quad | \quad s[w_1][c] \oplus \text{ans}\langle \text{"life"} \rangle . 0 \quad | \quad P_{w_2} \\
&\rightarrow 0 \quad | \quad P_s \quad | \quad 0 \quad | \quad P_{w_1} \quad | \quad 0 \quad | \quad P_{w_2} \\
&\equiv P_s \quad | \quad P_{w_1} \quad | \quad P_{w_2}
\end{aligned}$$

2.2 Types

Figure 4 shows the syntax of MPST! types.

Syntax of types. Session types S type communication endpoints. They consist of branching types $S^{\&}$, replicated branching types $!(S^{\&})$, selection types $S^{\oplus}$, recursive types $\mu t. S$ and variables t , and the **end** type.

Branching session types have the form $\rho \&_{i \in I} m_i(\tilde{T}_i).S_i$ indicating that role ρ offers a choice of message labels m_i with payload types $\tilde{T}_i$ and continuations S_i . Similarly, selection session types $\oplus_{i \in I} \rho_i m_i(\tilde{T}_i).S_i$ indicate an internal choice towards one of a set of roles ρ_i , with a message label, given payload types and continues as the corresponding continuation type. A replicated branching type $!\rho \&_{i \in I} m_i(\tilde{T}_i).S_i$ types a replicated channel. As with processes, role ρ can either be a concrete role, or a role variable in *binding* position.

Our type system supports *singleton* types for roles: role ρ has singleton type ρ , used to pattern-match specific roles in payloads. *Static* types T are used for protocol specification, whereas *runtime* types U are used by the type semantics to include a notion of *parallel composition* at type level: originally introduced by Le Brun and Dardha [4], a type $U_1 \mid U_2$ allows a channel to be associated with multiple “active” session types.

Session types	$S ::= S^{\&} \mid !(S^{\&}) \mid S^{\oplus} \mid \mu t.S \mid t \mid \mathbf{end}$	
Branching/Selection Types	$S^{\&} ::= \rho \&_{i \in I} m_i(\tilde{T}_i).S_i$	$S^{\oplus} ::= \oplus_{i \in I} \rho_i m_i(\tilde{T}_i).S_i$
Role singletons	$\rho ::= q \mid \alpha$	
Static types	$T ::= S \mid \rho$	
Runtime types	$U ::= S \mid (U_1 \mid U_2)$	

Fig. 4. Syntax of Types

We assume that branching and selection types include a non-empty set of messages with pairwise distinct message names m_i (per role for $\oplus$). We further take an *equi-recursive* view of types, identifying a recursive type with its unfolding (i.e., $\mu t.S = S\{\mu t.S/t\}$) and require that recursion variables are guarded.

Definition 2 (Subtyping). The *subtyping relation* $\leq$ is co-inductively defined on types by the following inference rules:

$$\begin{array}{c}
 \frac{(\tilde{T}_i \leq \tilde{T}'_i)_{i \in I} \quad (S_i \leq S'_i)_{i \in I}}{\rho \&_{i \in I} m_i(\tilde{T}_i).S_i \leq \rho \&_{i \in I \cup J} m_i(\tilde{T}'_i).S'_i} \quad \frac{(\tilde{T}_i \geq \tilde{T}'_i)_{i \in I} \quad (S_i \leq S'_i)_{i \in I}}{\oplus_{i \in I \cup J} \rho_i m_i(\tilde{T}_i).S_i \leq \oplus_{i \in I} \rho_i m_i(\tilde{T}'_i).S'_i} \\
 \\
 \frac{S_1^{\&} \leq S_2^{\&}}{!S_1^{\&} \leq !S_2^{\&}} \quad \frac{U \leq U' \quad S \leq S'}{U \mid S \leq U' \mid S'} \quad \frac{S\{\mu t.S/t\} \leq S'}{\mu t.S \leq S'} \quad \frac{S \leq S' \quad \{\mu t.S'/t\}}{S \leq \mu t.S'} \quad \frac{}{T \leq T}
 \end{array}$$

Our definition of subtyping (Definition 2) is mostly standard. We adopt the convention of smaller types being ones with less external choice and more internal choice (*à la* Gay and Hole [13]). Subtyping of replication is based on regular branching; and parallel types are related iff their session types are subtypes. Subtypes are related up-to their recursive unfolding, and subtyping is *reflexive*.

Definition 3 (Type Congruence). Type congruence allows us to treat parallel runtime types as commutative and associative with identity element **end**.

$$U_1 \mid U_2 \equiv U_2 \mid U_1 \quad (U_1 \mid U_2) \mid U_3 \equiv U_1 \mid (U_2 \mid U_3) \quad U \mid \mathbf{end} \equiv U$$

Figure 5 shows the definition of typing contexts and their operations. Context Θ is used to type recursive process definitions, mapping process variables X to tuples of parameter types. Context Γ maps channels to types, and role variable singletons. Context composition Γ, Γ' is defined iff Γ and Γ' have disjoint domains. We lift subtyping and type congruence to contexts in the usual way.

As inspired by (e.g.) [31], linearity is enforced through the use of a *context split* operation $\Gamma = \Gamma_1 \cdot \Gamma_2$ that splits a context Γ into two environments Γ_1 and Γ_2 . These environments may share variables with unrestricted types and role variables. Additionally, a channel c with runtime type $U_1 \mid U_2$ may be split such that Γ_1 contains $c : U_1$ and Γ_2 contains $c : U_2$; this allows us to type endpoints used by different replicated processes. The inverse operation is *context addition*

Typing contexts

$$\Theta ::= \emptyset \mid \Theta, X : \tilde{S} \quad \Gamma ::= \emptyset \mid \Gamma, c : U \mid \Gamma, \alpha : \alpha$$

Context splitting

$$\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$$

$$\begin{array}{c} \overline{\emptyset = \emptyset \cdot \emptyset} \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1, c : U \cdot \Gamma_2} \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1 \cdot \Gamma_2, c : U} \\[10pt] \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, \alpha : \alpha = \Gamma_1, \alpha : \alpha \cdot \Gamma_2, \alpha : \alpha} \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U_1 \mid U_2 = \Gamma_1, c : U_1 \cdot \Gamma_2, c : U_2} \end{array}$$

Context addition

$$\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$$

$$\begin{array}{c} \frac{}{\Gamma + \emptyset = \Gamma} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad a \notin \text{dom}(\Gamma_2)}{\Gamma_1, a : T + \Gamma_2 = \Gamma, a : T} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad a \notin \text{dom}(\Gamma_1)}{\Gamma_1 + \Gamma_2, a : T = \Gamma, a : T} \\[10pt] \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, \alpha : \alpha + \Gamma_2, \alpha : \alpha = \Gamma, \alpha : \alpha} \quad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, c : U_1 + \Gamma_2, c : U_2 = \Gamma, c : U_1 \mid U_2} \end{array}$$

Context role insertion

$$\boxed{\Gamma \leftarrow \rho}$$

$$\Gamma \leftarrow q = \Gamma \quad \Gamma \leftarrow \alpha = \Gamma + \alpha : \alpha$$

Fig. 5. Typing contexts and context operations.

$\Gamma_1 + \Gamma_2 = \Gamma$ that combines Γ_1 and Γ_2 into an environment Γ ; again roles may be shared across environments. In the case that we have $\Gamma_1, c : U_1 + \Gamma_2, c : U_2$ (i.e., a linear variable used in two different contexts), context addition results in c having the combined runtime type $U_1 \mid U_2$. The *role insertion* operation $\Gamma \leftarrow \rho$ is used when typing replicated receives and extends a context with a mapping $\alpha : \alpha$ in the case that ρ is a role variable, and leaves Γ unchanged otherwise.

Before presenting the typing rules, we introduce the notion of a *session protocol*, mapping role names to session types. The $\text{assoc}_s(\Psi)$ function associates a session protocol Ψ to a concrete session s to form a typing context.

Definition 4 (Session Protocol). A session protocol Ψ is attached to a session through the restriction operation in the form of $(\nu s : \Psi) P$, dictating the protocol for s in P . A protocol Ψ is a partial mapping from role to session type, given as:

$$\Psi ::= \emptyset \mid \Psi, q : S$$

A protocol Ψ is well-formed iff it does not contain parallel types. We obtain a typing context by associating roles in a protocol with a session name. Formally:

$$\text{assoc}_s(q : S, \Psi) := s[q] : S, \text{assoc}_s(\Psi) \quad \text{assoc}_s(\emptyset) := \emptyset$$

Typing Rules for Values and Recursive Definitions

$$\boxed{\Gamma \vdash V : T} \quad \boxed{\Theta \vdash X : \tilde{S}}$$

T-WKN

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \text{end}(\Gamma_2)}{\Gamma \vdash V : T}$$

$$\frac{\text{T-}q}{\emptyset \vdash q : q}$$

$$\frac{\text{T-SUB} \quad S \leq S'}{c : S \vdash c : S'}$$

$$\frac{\text{T-}\alpha}{\alpha : \alpha \vdash \alpha : \alpha}$$

$$\frac{\text{T-X} \quad \Theta(X) = (S_i)_{i \in 1..n}}{\Theta \vdash X : (S_i)_{i \in 1..n}}$$

Typing Rules for Processes

$$\boxed{\Theta; \Gamma \vdash P}$$

$$\begin{array}{c} \text{T-0} \\ \text{end}(\Gamma) \\ \hline \Theta; \Gamma \vdash 0 \end{array} \quad \begin{array}{c} \text{T-}| \\ \Theta; \Gamma_1 \vdash P_1 \quad \Theta; \Gamma_2 \vdash P_2 \\ \hline \Theta; \Gamma_1 \cdot \Gamma_2 \vdash P_1 | P_2 \end{array} \quad \begin{array}{c} \text{T-+} \\ (\Theta; \Gamma \vdash c_i[\rho_i] \oplus m_i\langle \tilde{d}_i \rangle \cdot P_i)_{i \in I} \\ \hline \Theta; \Gamma \vdash \sum_{i \in I} c_i[\rho_i] \oplus m_i\langle \tilde{d}_i \rangle \cdot P_i \end{array}$$

$$\begin{array}{c} \text{T-!} \\ \Gamma_1 \vdash c : !\rho \&_{i \in I} m_i(\tilde{T}_i).S'_i \quad \text{end}(\Gamma) \\ (\Theta; \Gamma + c : S'_i + \tilde{b}_i : \tilde{T}_i \leftarrow \rho \vdash P_i)_{i \in I} \\ \hline \Theta; \Gamma \cdot \Gamma_1 \vdash !c[\rho] \&_{i \in I} m_i(\tilde{b}_i) \cdot P_i \end{array} \quad \begin{array}{c} \text{T-}\& \\ \Gamma_{\&} \vdash c : \rho \&_{i \in I} m_i(\tilde{T}_i).S'_i \quad \Gamma_r \vdash \rho : \rho \\ (\Theta; \Gamma + c : S'_i + \tilde{b}_i : \tilde{T}_i \vdash P_i)_{i \in I} \\ \hline \Theta; \Gamma \cdot \Gamma_{\&} \cdot \Gamma_r \vdash c[\rho] \&_{i \in I} m_i(\tilde{b}_i) \cdot P_i \end{array}$$

T-ν

$$\begin{array}{c} \varphi(\text{assoc}_s(\Psi)) \\ \varphi \text{ is a safety property} \\ s \notin \Gamma \quad \Theta; \Gamma + \text{assoc}_s(\Psi) \vdash P \\ \hline \Theta; \Gamma \vdash (\nu s : \Psi) P \end{array} \quad \begin{array}{c} \text{T-}\oplus \\ \Gamma_{\oplus} \vdash c : \rho \oplus m(\tilde{T}).S' \quad \Gamma_r \vdash \rho : \rho \\ (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Theta; \Gamma + c : S' \vdash P \\ \hline \Theta; \Gamma \cdot \Gamma_{\oplus} \cdot \Gamma_r(\cdot \Gamma_i)_{i \in 1..n} \vdash c[\rho] \oplus m((V_i)_{i \in 1..n}) \cdot P \end{array}$$

T-DEF

$$\frac{\Theta, X : \tilde{S}; \Gamma, x : \tilde{S} \vdash P \quad \text{end}(\Gamma) \quad \Theta, X : \tilde{S}; \Gamma' \vdash Q}{\Theta; \Gamma \cdot \Gamma' \vdash \text{def } X(\tilde{x} : \tilde{S}) = P \text{ in } Q}$$

T-CALL

$$\frac{\Theta \vdash X : (S_i)_{i \in 1..n} \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n : \Gamma_i \vdash c_i : S_i}{\Theta; \Gamma_0(\cdot \Gamma_i)_{i \in 1..n} \vdash X((c_i)_{i \in 1..n})}$$

Fig. 6. Typing rules.

Next, we introduce predicate end on type contexts. This ensures that an environment is *end-typed*: that is, its session types are congruent to type **end**.

Definition 5 (End-typed environment). A context is end-typed, written $\text{end}(\Gamma)$, iff it only maps channels to session type **end**.

$$\frac{}{\text{end}(\emptyset)} \quad \frac{\text{end}(\Gamma)}{\text{end}(c : \mathbf{end}, \Gamma)} \quad \frac{\text{end}(\Gamma)}{\text{end}(\alpha : \alpha, \Gamma)} \quad \frac{\Gamma \equiv \Gamma' \quad \text{end}(\Gamma)}{\text{end}(\Gamma')}$$

Figure 6 shows the typing rules for MPST!. There are three judgements: the value typing judgement $\Gamma \vdash V : T$ assigns type T to value V under context Γ and consists of four rules. Rule [T-Wkn] allows weakening: if value V has type T under some environment Γ_1 , and we have an end-typed environment Γ_2 such that $\Gamma_1 + \Gamma_2 = \Gamma$, then the rule allows us to conclude that V has type T under Γ . Rule [T- q] types concrete roles as singleton types under the empty environment, whereas [T- α] types a role variable provided that it is contained within the type

environment. Finally, rule [T-Sub] types a linear variable, allowing for subtyping. Judgement $\Theta \vdash X : \tilde{S}$ simply looks up process variable X in process environment Θ , returning its parameter types; this is achieved by rule [T-X].

The final judgement $\Theta ; \Gamma \vdash P$ states that process P is typable under recursion environment Θ and type environment Γ . Rule [T-0] types the inactive process under an end-typed environment. Rule [T-|] types parallel processes under a split environment. Rule [T-+] types an output-directed choice; since this operation uses branching control flow, each choice must be typable under the same environment.

Rules [T-!] and [T-&] type replicated and non-replicated receives respectively. In both cases the rules check that the channel has a receiving session type. Both rules check that each branch is typable by extending a common typing context with the variables bound by the receives, along with the continuation type for the session channel. In the case of a replicated receive there are two differences: first, the context used to type each branch must be end-typed (in order to avoid duplicating linear resources). Second, since the role ρ may be a *binding* occurrence of a role variable α , the context used to type each branch must be extended with the role variable (if applicable) using the insertion operator.

Rule [T- ν] types a session name restriction $(\nu s : \Psi)P$. As is standard in generalised MPST [27], the session protocol must satisfy a *safety property* φ . We discuss specifics of this property in Section 2.3, and how it is used in Section 2.4. Informally, the property ensures that all process communication is “compatible”, and is the weakest property required to prove subject reduction. We then prove our metatheory parametric on the *largest* safety property, allowing it to be customised to verify more precise properties (e.g., to ensure deadlock-freedom or termination), based on the requirements of a specific protocol.

Rule [T- $\oplus$] types an output if the sending channel can be mapped to a selection type with payload types that match the values being sent. The rule ensures that role ρ is either a value, or it exists in the type context—*i.e.*, messages cannot be sent to unbound role variables. The selection type continuation should be used, along with the common context, to type the continuation process.

Lastly, rules [T-Def] and [T-Call] type recursive processes. The former populates the recursion environment whilst ensuring that process declarations are well typed under the variables they bind. The latter checks that types of values used in a process call match what was specified in the declaration.

Example 4 (Load Balancer: Type Checking). By introducing a name restriction for session s that includes the types from Example 2, we can type the processes from Example 3 under empty typing contexts:

$$\emptyset; \emptyset \vdash (\nu s : \{s : S_s, w_1 : S_{w_1}, w_2 : S_{w_2}, c : S_c\}) P_s \mid P_{w_1} \mid P_{w_2} \mid P_c$$

2.3 Type Semantics

To reason about the interactions between session types, following Scalas & Yoshida [27], we endow typing contexts Γ with LTS semantics as shown in Figure 7. Each action γ denotes an *output*, *input*, and *synchronising communication* respectively. *Context reduction* $\Gamma \rightarrow \Gamma'$ is defined iff $\Gamma \xrightarrow{s:p, q:m} \Gamma'$ for some

Actions $\gamma ::= s:\textcolor{blue}{q} \oplus \textcolor{red}{r}:\text{m}(\tilde{T}) \mid s:\textcolor{blue}{q} \& \textcolor{red}{r}:\text{m}(\tilde{T}) \mid s:\textcolor{blue}{q}, \textcolor{red}{r}:\text{m}$

Context Reduction

$\Gamma \rightarrow \Gamma'$

$$\begin{array}{c}
\text{Γ-}\& \quad \text{Γ-}\oplus \quad \text{Γ-}\rho \\
\frac{S = \mathbf{q}\&_{i \in I} \mathbf{m}_i(\tilde{T}_i).S'_i \quad k \in I}{s[\mathbf{p}] : S \xrightarrow{s:\mathbf{p}\&\mathbf{q}:\mathbf{m}_k(\tilde{T}_k)} s[\mathbf{p}] : S'_k} \quad \frac{S = \oplus_{i \in I} \mathbf{q}_i \mathbf{m}_i(\tilde{T}_i).S'_i \quad k \in I}{s[\mathbf{p}] : S \xrightarrow{s:\mathbf{p}\oplus\mathbf{q}_k:\mathbf{m}_k(\tilde{T}_k)} s[\mathbf{p}] : S'_k} \quad \frac{\Gamma \xrightarrow{\gamma} \Gamma'}{\Gamma, \rho : \rho \xrightarrow{\gamma} \Gamma', \rho : \rho} \\
\\
\text{Γ-!} \quad \text{Γ-CONG} \quad \text{Γ-}\mu \\
\frac{R = !\rho\&_{i \in I} \mathbf{m}_i(\tilde{T}_i).S_i \quad k \in I}{s[\mathbf{q}] : R \xrightarrow{s:\mathbf{q}\&\rho:\mathbf{m}_k(\tilde{T}_k)} s[\mathbf{q}] : R \mid S_k} \quad \frac{\Gamma \xrightarrow{\gamma} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{\gamma} \Gamma' + \Gamma''} \quad \frac{\Gamma \xrightarrow{\mu} \Gamma'}{\Gamma \cdot c : S\{\mu t.S/t\} \xrightarrow{\gamma} \Gamma'} \\
\\
\text{Γ-COM}_1 \\
\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \xrightarrow{s:\mathbf{p}\oplus\mathbf{q}:\mathbf{m}(\tilde{S}, \tilde{\mathbf{r}})} \Gamma'_1 \quad \Gamma_2 \xrightarrow{s:\mathbf{q}\&\mathbf{p}:\mathbf{m}(\tilde{S}', \tilde{\alpha})} \Gamma'_2 \quad \tilde{S} \leq \tilde{S}'}{\Gamma \xrightarrow{s:\mathbf{p}, \mathbf{q}:\mathbf{m}} \Gamma'_1 + \Gamma'_2\{\tilde{\mathbf{r}}/\tilde{\alpha}\}} \\
\\
\text{Γ-COM}_2 \\
\frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \quad \Gamma_1 \xrightarrow{s:\mathbf{p}\oplus\mathbf{q}:\mathbf{m}(\tilde{S}, \tilde{\mathbf{r}})} \Gamma'_1 \quad \Gamma_2 \xrightarrow{s:\mathbf{q}\&\alpha_0:\mathbf{m}(\tilde{S}', \tilde{\alpha})} \Gamma'_2 \quad \tilde{S} \leq \tilde{S}'}{\Gamma \xrightarrow{s:\mathbf{p}, \mathbf{q}:\mathbf{m}} \Gamma'_1 + \Gamma'_2\{\mathbf{p}/\alpha_0\}\{\tilde{\mathbf{r}}/\tilde{\alpha}\}}
\end{array}$$

Fig. 7. Type semantics.

$s, \mathbf{p}, \mathbf{q}, \mathbf{m}$, and we write $\rightarrow^*$ for the transitive and reflexive closure of $\rightarrow$. We write $\Gamma \rightarrow$ iff $\Gamma \rightarrow \Gamma'$ for some Γ' .

Transitions $[\Gamma\text{-}\&]$ and $[\Gamma\text{-}\oplus]$ are standard: a context can fire an input label (resp. output label) matching any of the labels in the top-level branch type (resp. selection type). This transitions the entry to the continuation of the chosen type.

Transition $[T-!]$ models the receipt of a message by a replicated input. The two main differences to the linear receive are: (i) the role ρ used in the transition label is allowed to be a role variable name; and (ii) firing an input does not advance the type, but instead pulls out a copy of the continuation and places it in parallel. Role ρ is considered bound in R and its continuations, but is *free* in pulled out copies of the continuations composed in parallel (S_k).

Transition rules $[I\text{-}\rho]$, $[I\text{-}\text{Cong}]$, $[I\text{-}\mu]$ allow contexts to reduce under a larger context, or when types are guarded by recursive binders. Concretely, $[I\text{-}\rho]$ allows transitions to ignore role singletons; $[I\text{-}\text{Cong}]$ allows a context to perform a transition when split from a larger context (the transition result must be added back in); and $[I\text{-}\mu]$ allows recursive binders to be treated equal to their unfolding.

Transitions $[\Gamma\text{-Com}_1]$ and $[\Gamma\text{-Com}_2]$ model type-level communication; for simplicity and without loss of generality we assume a convention wherein session-typed payloads precede role-typed payloads. Both rules state that if a context can be split such that one part fires an output label, and the other fires an input with matching roles, message label and payloads, then the entire context can transition

via a *communication* action. We note that payloads will consist of either session types or role singletons. For the former, sender payloads must be subtypes of the receiver payloads; for the latter, *role substitution* occurs after communication. $[\Gamma\text{-Com}_2]$ caters for universal receives, where the input label consists of a role variable in binding position—this is reflected in the role substitution. We say that a context *reduces* iff it can transition via communication actions.

Example 5 (Load Balancer: Context Reduction). We now use the protocol from Example 4 to demonstrate context reduction in action. Initially, the only communication action possible is between the client and server, via $[\Gamma\text{-Com}_2]$.

$$\begin{aligned}
& s[c]:S_c, s[s]:S_s, s[w_1]:S_{w_1}, s[w_2]:S_{w_2} \\
& = \\
& (s[c]:S_c \cdot s[s]:S_s) \cdot (s[w_1]:S_{w_1}, s[w_2]:S_{w_2}) \\
& \rightarrow \\
& s[c]:s\&wrk(\omega).\omega\&ans(\text{str}) + \left(s[s]:S_s \mid \oplus \left\{ \begin{array}{l} w_1\text{fw}(\text{int}, \alpha).\alpha\oplus wrk(w_1) \\ w_2\text{fw}(\text{int}, \alpha).\alpha\oplus wrk(w_2) \end{array} \right\} \{c/\alpha\} \right) \\
& + s[w_1]:S_{w_1}, s[w_2]:S_{w_2} \\
& = \\
& s[c]:s\&wrk(\omega).\omega\&ans(\text{str}), \\
& s[s]:S_s \mid \oplus \left\{ \begin{array}{l} w_1\text{fw}(\text{int}, c).c\oplus wrk(w_1) \\ w_2\text{fw}(\text{int}, c).c\oplus wrk(w_2) \end{array} \right\}, s[w_1]:S_{w_1}, s[w_2]:S_{w_2}
\end{aligned}$$

It is key to note the role substitution for the type of $s[s]$ above; specifically, how the substitution affects the parallel type extracted through communication, but does not affect the replicated type S_s . From here, there are multiple reduction paths, but let us observe the reduction path in which s communicates with w_1 .

$$\begin{aligned}
& = \\
& s[c]:s\&wrk(\omega).\omega\&ans(\text{str}), s[w_2]:S_{w_2}, s[s]:S_s \\
& \cdot \left(s[s]:\oplus \left\{ \begin{array}{l} w_1\text{fw}(\text{int}, c).c\oplus wrk(w_1) \\ w_2\text{fw}(\text{int}, c).c\oplus wrk(w_2) \end{array} \right\} \cdot s[w_1]:S_{w_1} \right) \\
& \rightarrow \\
& s[c]:s\&wrk(\omega).\omega\&ans(\text{str}), s[w_2]:S_{w_2}, s[s]:S_s \\
& + (s[s]:c\oplus wrk(w_1) + (s[w_1]:S_{w_1} \mid \gamma\oplus ans(\text{str}))\{c/\gamma\}) \\
& = \\
& s[c]:s\&wrk(\omega).\omega\&ans(\text{str}), s[w_2]:S_{w_2}, \\
& s[s]:S_s \mid c\oplus wrk(w_1), s[w_1]:S_{w_1} \mid c\oplus ans(\text{str})
\end{aligned}$$

Note how reduction is possible because the context split allows the server's parallel type to be extracted into its own context. Then, reduction occurs via $[\Gamma\text{-Cong}]$ and $[\Gamma\text{-Com}_1]$. From here, the context reduces in a similar fashion: first the server communicates a role with the client; followed by the final communication between the client and worker.

$$\begin{aligned}
& \rightarrow \\
& s[c]:w_1\&ans(\text{str}), s[s]:S_s \mid \text{end}, s[w_2]:S_{w_2}, s[w_1]:S_{w_1} \mid c\oplus ans(\text{str}) \\
& \rightarrow \\
& s[c]:\text{end}, s[s]:S_s \mid \text{end}, s[w_2]:S_{w_2}, s[w_1]:S_{w_1} \mid \text{end} \\
& \equiv \\
& s[c]:\text{end}, s[s]:S_s, s[w_2]:S_{w_2}, s[w_1]:S_{w_1}
\end{aligned}$$

Safety. In order to type a session restriction, rule [T- ν] in Figure 6 requires that the session’s protocol of the session must obey a *safety property* φ . Informally, a safety property requires that processes exchange values of compatible types and that a sender only ever selects available branches. Safety is the weakest typing context property required in order to prove subject reduction.

Definition 6 (Safety). φ is a safety property on type environment Γ iff:

$$\begin{array}{l}
 \text{S-}\oplus\& \\
 \varphi \left(\Gamma \cdot s[p] : \oplus_{i \in I} \rho_i \mathbf{m}_i(\tilde{S}_i, \tilde{\mathbf{r}}_i).S'_i \cdot s[q] : p \&_{j \in J} \mathbf{m}_j(\tilde{S}''_j, \tilde{\alpha}_j).S'''_j \right) \\
 \text{and } \exists K \subseteq I \text{ s.t. } \forall k \in K : \rho_k = \mathbf{q} \\
 \text{implies } K \subseteq J \text{ and } \forall i \in K : \tilde{S}_i \leq \tilde{S}''_i \text{ and } |\tilde{\mathbf{r}}_i| = |\tilde{\alpha}_i| \\
 \\
 \text{S-}!\oplus\& \\
 \varphi \left(\Gamma \cdot s[p] : \oplus_{i \in I} \rho_i \mathbf{m}_i(\tilde{S}_i, \tilde{\mathbf{r}}_i).S'_i \cdot s[q] : !\rho_0 \&_{j \in J} \mathbf{m}_j(\tilde{S}''_j, \tilde{\alpha}_j).S'''_j \right) \\
 \text{and } \exists K \subseteq I \text{ s.t. } \forall k \in K : \rho_k = \mathbf{q} \text{ and } \rho_0 \text{ is either a variable or } p \\
 \text{implies } K \subseteq J \text{ and } \forall i \in K : \tilde{S}_i \leq \tilde{S}''_i \text{ and } |\tilde{\mathbf{r}}_i| = |\tilde{\alpha}_i| \\
 \\
 \text{S-}\mu \\
 \varphi(\Gamma \cdot s[q] : \mu t.S) \text{ implies } \varphi(\Gamma \cdot s[q] : S\{\mu t.S/t\}) \\
 \\
 \text{S-}\alpha \quad \varphi(\Gamma) \text{ implies } \text{frv}(\Gamma) = \emptyset \quad \text{S-}\rightarrow \quad \varphi(\Gamma) \text{ and } \Gamma \rightarrow \Gamma' \text{ implies } \varphi(\Gamma')
 \end{array}$$

A property φ is considered *safe* iff it conforms to all conditions specified in Definition 6. Conditions [S- $\oplus\&$] and [S- $!\oplus\&$] are concerned with communication. They state that if two roles in the same session have an output and input type respectively which point at each other, then: (i) they should have at least one common message label; (ii) for each common label, their payloads should be equal in length; and (iii) for each common label, all session types in the payload of the sender should be subtypes of what is expected by the receiver.

Condition [S- μ] requires safety to hold after the unfolding of recursive binders; [S- α] requires all role variables used in a context to be bound by that same context; and [S- $\rightarrow$] requires safety to hold after context reduction.

Users of the type system can re-instantiate φ with custom properties (*e.g.*, *termination*), as long as the property used meets the safety requirements.

2.4 Metatheory

Unlike most session type theories, *generalised* MPST do not syntactically guarantee any properties on the processes they type. Rather, they provide a framework for verifying runtime properties on the type context, from which process-level properties may be inferred—the benefit being that these properties are undecidable to check on processes, yet decidable in the realm of the type system. Furthermore, its generalised nature allows for fine-tuning based on specific requirements of its applications. Informally, generalisation of the type system works by proving the metatheory parametric of the largest safety property captured

by φ in Definition 6; *i.e.*, all theorems proved and presented which involve a type context Γ , assume that $\varphi(\Gamma)$ holds, or “ Γ is safe”. Essentially, φ describes the minimum (and most general) *safety* requirements made for *subject reduction* to hold. This proof technique allows φ to be re-instantiated with more specific properties without having to reprove any of the base metatheory. (We occasionally reference properties other than *safety* in examples.)

2.5 Subject Reduction and Session Fidelity

The main results of a generalised MPST system are *subject reduction* (Theorem 1) and *session fidelity* (Theorem 2). These theorems allow the type system to be used as a framework for verifying custom properties by re-instantiating φ . We note that discussing *how* the generalised type system can be used as a verification framework is *not* the focus of this paper (to this end, we address the interested reader to Scalas and Yoshida [27, Section 5]); rather, we build on generalised MPST theory as a means of investigating the expressiveness of replication and first-class roles in MPST. Hence, the following presents the two theorems—highlighting key differences with what is standard—but we do not demonstrate the verification of runtime properties (which is standard). Proofs are in the technical report [20].

Theorem 1 (Subject Reduction). *If $\Theta; \Gamma \vdash P$ with $\varphi(\Gamma)$ and $P \rightarrow P'$, then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Theta; \Gamma' \vdash P'$ with $\varphi(\Gamma')$.*

Intuitively, subject reduction states that any *safe* and *well-typed* process remains safe and well-typed after process reduction. More formally, the theorem asserts that if a process is typed under a safe context, then the context can match any process reduction to type its continuation whilst retaining safety.

Session fidelity states that if a context can reduce, then a process it types can observe *at least one* of its reductions. By virtue of subtyping, a context is allowed to specify paths which need not be followed by a process it types; the key point is that session fidelity requires that there is *at least one* observable reduction. Using session fidelity, one can prove properties about communication occurring within a single session. It does not, however, provide grounds for showing such properties on interleaved session communication. Hence, as is standard in generalised MPST, we define additional assumptions on processes required for session fidelity to hold.

Definition 7 (Only plays one role). *(The following is a slight adaptation of [27, definition 5.3].) Assuming $\emptyset; \Gamma \vdash P$, we say P :*

1. *has guarded definitions* iff each subterm of P with the form

$$\text{def } X((x_i : S_i)_{i \in 1..n}) = Q \text{ in } P'$$

we have: $\forall i \in 1..n : S_i \not\leq \text{end}$ implies a process call $Y\langle \dots, x_i, \dots \rangle$ can only occur in Q as a subterm of a communication action over channel x_i .

2. *only plays role $\mathbf{p}$ in s , by Γ* iff (i) Item 1 holds for P ; (ii) $\text{fv}(P) = \emptyset$; (iii) $\Gamma = \Gamma_0, s[\mathbf{p}]:U$ with $U \not\leq \text{end}$ and $\text{end}(\Gamma_0)$; (iv) in all subterms $(\nu s : \Gamma') P'$ of P , we have $\text{end}(\Gamma')$.

The purpose of Item 1 is to prevent processes from infinitely reducing via [R-X] without communicating, and Item 2 identifies a typical application of MPST where a number of processes P_p communicate over a multiparty session s , with each process playing a *single* role. The difference in our definition to the standard is that processes P_p should play a *single* role and not a *unique* role. This is due to the introduction of replication in our language; note how reduction with a replicated process is guaranteed to produce multiple processes playing the same role and is reflected in our definition in condition (iii) of Item 2, where a channel is mapped to *runtime type* U , allowing for parallel composition.

Informally, the session fidelity theorem states that, given a safe context that types a process of a particular structure—*i.e.*, one governed by the session fidelity assumptions of Definition 7—then if the context can reduce, the typed process can match at least one reduction. Furthermore, after the process matches the context reduction, it remains within the session fidelity assumption structure.

Theorem 2 (Session Fidelity). *Assume $\emptyset; \Gamma \vdash P$ with $\varphi(\Gamma)$ and $P \equiv \prod_{p \in I} P_p$ where each P_p is either $\mathbf{0}$ (up-to $\equiv$), or only plays role p in s . Then, $\Gamma \rightarrow$ implies $\exists \Gamma', P'$ s.t. $\Gamma \rightarrow \Gamma'$, $P \rightarrow^* P'$ and $\Gamma' \vdash P'$, where $P' \equiv \prod_{p \in I} P'_p$ and each P'_p is either $\mathbf{0}$ (up-to $\equiv$), or only plays role p in s .*

We now turn our attention to the main focus of this paper, *i.e.*, exploring the expressiveness and decidability of replication and first-class roles in MPST.

3 Expressivity and Decidability

This section discusses, and shows by example, the benefits and limitations of MPST!. Section 3.1 demonstrates the expressivity gained by using replication and first-class roles in MPST, whilst Section 3.2 presents our decidability results.

3.1 Expressivity

We begin by demonstrating a common design pattern used for describing protocols, which we call *services*. We build a number of services to showcase language features, and to describe protocols which—to the best of our knowledge—were previously untypable in any MPST theory. Specifically, using the increased expressiveness of replication and first-class roles, we define types for *binary trees*, the *dining philosophers problem*, and an *auction*. Importantly, all examples shown adhere to the decidability requirements discussed later (*cf.* Section 3.2).

Services. A *service* is a building-block of a protocol, involving some universal receive, with the aim of *offering* a specific interaction. A *client* interacts with a service to achieve the communication pattern it offers. Importantly, services may be clients of other services, promoting a modular design of protocols in MPST!.

Example 6 (Ping). The *ping service* simply responds to a ping with a pong.

$P : !\alpha \& \text{ping} . \alpha \oplus \text{pong} . \text{end}$

A basic yet useful service is given in Example 6, where role P offers a *ping* service. As a convention, we will use capitals for naming services. We highlight the importance of both replication and free role names in types to be able to design modular components of a protocol—both are integral to designing a sub-protocol agnostic of the larger scope in which it is used.

Context-Free MPST. Context-free session types [29,1,22] are a formalism that replace prefix-style session types with individual actions, along with a sequencing operator $;$ with neutral element *skip*. The goal of this line of work is to express communication protocols that are not possible under tail-recursive session types, given their restriction to regular languages. The classic example is that of communicating a serialised binary tree.

Example 7 (Binary tree in standard context-free STs). Consider a binary tree data type described by the following context-free grammar.

$$\text{tree} ::= (\text{node}, \text{tree}, \text{tree}) \mid \text{leaf}$$

We could attempt to represent a protocol that serialises this data type as follows:

$$\mu t. \oplus \{\text{leaf} : \text{skip}, \text{node} : t\}$$

However, this type is not precise enough—it does not guarantee that the correct structure of a binary tree is maintained. Work on context-free session types solves this by proposing type systems allowing the following specification:

$$\mu t. \oplus \{\text{leaf} : \text{skip}, \text{node} : \text{skip}; t; t\}$$

Selecting the *node* label now guarantees that *two* sub-trees will follow.

The parallel types presented in Section 2.2, although not exposed directly to users, lift expressiveness of types in MPST!. In fact, since replicated branches are *permanently available* (by composing continuation types in parallel), we can simulate the sequencing operator $;$ using type-level parallel composition.

Example 8 (Binary Tree Service). Recall the ping service P from Example 6. We build a *binary tree service* T using P as shown below:

$$T : !\beta \& \text{tree}. P \oplus \text{ping}. !P \& \text{pong}. \beta \& \{\text{leaf}. \text{end}, \text{node}. P \oplus \text{ping}. P \oplus \text{ping}. \text{end}\}$$

The service begins by receiving a request for a *tree* from a client. It then sends a *ping* to the ping service, exposing a replicated branch waiting to receive the *pong* reply. The client is now free to build the binary tree. It is key to note that any *node* sent to the service will subsequently forward *two* *ping* requests to P . In turn, this communication will pull out two copies of the type continuation $\beta \& \{\text{leaf}. \text{end}, \text{node}. P \oplus \text{ping}. P \oplus \text{ping}. \text{end}\}$, forcing the client to maintain the appropriate binary tree structure. For example, if a client t wishes to build a tree consisting of one root node and two leaf nodes, its type would be defined as:

$$t : T \oplus \text{tree}. T \oplus \text{node}. T \oplus \text{leaf}. T \oplus \text{leaf}. \text{end}$$

The metatheoretic framework can be used to determine that any protocol failing to abide by the binary tree structure will result in a *deadlock*; whilst any safe protocol that obeys the correct structure is *terminating*, e.g. the protocol $\{t, T, P\}$.

An obscure limitation of the tree service in Example 8 is that it can only be used by a single client. Consider, for example, two separate clients sending a **node** message to **T**. Since both tree service types communicate with **P** to unroll the replicated branch **!P&pong**, the protocol becomes non-deterministic in a non-confluent manner and can result in deadlocked behaviour. To resolve this issue, we amend the tree service to accept a payload role which should act as a personal ping service for the client; this guarantees that the tree type is only unrolled by the client that made the initial request.

Example 9 (Multi-Client Binary Tree). We now redesign the binary tree service, this time capable of concurrently building multiple trees for different clients. The key difference here being that the new service, **M**, accepts a role as a payload on the initial request to which it will issue its **pings**.

$$M : !\alpha \& \text{tree}(\beta) . \beta \oplus \text{ping} . !\beta \& \text{pong} . \alpha \& \{ \text{leaf} . \text{end}, \text{node} . \beta \oplus \text{ping} . \beta \oplus \text{ping} . \text{end} \}$$

$$S_p = !M \& \text{ping} . M \oplus \text{pong} . \text{end}$$

Multiple clients can now issue concurrent requests to the tree service whilst maintaining safety. A sample (terminating) protocol is that of $\{t_1, t_2, p_1, p_2, M\}$, where $p_1, p_2 : S_p$, and the types for t_1, t_2 are given by:

$$t_1 : M \oplus \text{tree}(p_1) . M \oplus \text{node} . M \oplus \text{leaf} . M \oplus \text{leaf} . \text{end}$$

$$t_2 : M \oplus \text{tree}(p_2) . M \oplus \text{node} . M \oplus \text{leaf} . M \oplus \text{node} . M \oplus \text{leaf} . M \oplus \text{leaf} . \text{end}$$

Replication vs. Recursion. We have seen that replication and parallel composition increases the expressive power of MPST beyond that of tail-recursion. Naturally, one might ask, “*is recursion still needed?*” We find replication and recursion in MPST to be mutually non-inclusive—*i.e.*, both can produce protocols which *cannot* be typed under the other construct. We have already demonstrated this in one direction with the binary tree examples; below we showcase how recursion cannot be replaced by replication.

Example 10 (Lock Service). The lock service provides clients with a *mutex lock*.

$$L : !\theta \& \text{lock} . \mu t . \theta \& \{ \text{acquire} . \theta \& \text{release} . t, \text{done} . \text{end} \}$$

When a client requests a lock from **L**, a copy of the recursive continuation is exposed. The recursive definition allows sequences of **acquire** and **release** messages to be received. It is key to note that, whilst replication maintains a top-level branch that is permanently available to receive a message, the top-level action in a recursive definition is *not* fixed.

Copies of the continuation type of a replicated receive are executed concurrently. Example 10 provides a service for roles to enter race-sensitive portions of a protocol, as if it were an atomic action. We demonstrate its use by typing the dining philosophers problem.

Example 11 (Dining Philosophers). A number of philosophers gather to eat on a round table. Each plate is separated by a single chopstick, and a philosopher needs two chopsticks to eat. The dining philosophers problem requires the philosophers to employ an algorithm to ensure the table does not get deadlocked. In such a setting, we can view *chopsticks as services* and *philosophers as clients*. Assuming a size of n -philosophers, we define the type for a chopstick as:

$$(C_i)_{1..n} : L \oplus \text{lock} . !\eta \& \left\{ \begin{array}{l} \text{take?} . L \oplus \text{acquire} . \eta \oplus \text{ok} . \eta \& \text{give} . L \oplus \text{release} . \text{end} \\ \text{done} . L \oplus \text{done} . \text{end} \end{array} \right\}$$

Before offering its service, a chopstick requests a lock from L . This ensures that every chopstick has its own lock that it may **acquire** and **release**. The lock is used to guarantee that a chopstick is only ever taken by a single philosopher at a time. A chopstick then waits for a **take?** request from a philosopher; receiving one will result in it attempting to **acquire** the lock. This acquisition is only successful if the same role has not already requested it in some other parallel composition. If the lock was already acquired, then the $L \oplus \text{acquire}$ will block until the lock is released. Acquiring the lock sends an **ok** back to the philosopher, symbolising that they have successfully obtained the chopstick. When done from eating, the philosopher may then send back a **give** message, which in turn releases the lock, as the chopstick is now available to be taken by a different philosopher.

We can now write an algorithm for philosophers. First, a naive approach:

$$(p_i)_{i \in 1..n} : C_i \oplus \text{take?} . C_{i+1} \oplus \text{take?} . C_i \& \text{ok} . C_{i+1} \& \text{ok} . C_i \oplus \text{give} . C_{i+1} \oplus \text{give} . q \oplus \text{fin} . \text{end} \\ q : p_1 \& \text{fin} . \dots . p_n \& \text{fin} . C_1 \oplus \text{done} . \dots . C_n \oplus \text{done} . \text{end}$$

Every philosopher p_i has a similar type. They begin by requesting to take the chopsticks to their left and right—note that this results in every chopstick receiving two **take?** requests. Receiving both **ok** messages means the philosopher can eat, and subsequently **give** back the chopsticks. Finally, when finished, philosophers send a **fin** to role q , acting as a clean-up for the protocol. The protocol $\{p_i, q, C_i, L\}_{i \in 1..n}$ is safe, but fails typechecking for $\varphi = \text{terminating}$. In fact, the naïve protocol allows for scenarios in which all philosophers take a single chopstick, resulting in a deadlock. This problem has many solutions; we present the simplest in which philosophers take turns to eat. (Key changes are underlined.)

$$S_1 = C_1 \oplus \text{take?} . C_2 \oplus \text{take?} . C_1 \& \text{ok} . C_2 \& \text{ok} . C_1 \oplus \text{give} . C_2 \oplus \text{give} . \underline{p_2 \oplus \text{fin} . \text{end}} \\ S_2 = \underline{p_{i-1} \& \text{fin} . C_i \oplus \text{take?} . C_{i+1} \oplus \text{take?} . C_i \& \text{ok} . C_{i+1} \& \text{ok} . C_i \oplus \text{give} . C_{i+1} \oplus \text{give} . \underline{p_{i+1} \oplus \text{fin} . \text{end}}} \\ S_3 = \underline{p_{n-1} \& \text{fin} . C_n \oplus \text{take?} . C_1 \oplus \text{take?} . C_n \& \text{ok} . C_1 \& \text{ok} . C_n \oplus \text{give} . C_1 \oplus \text{give} . \underline{q \oplus \text{fin} . \text{end}}} \\ q' : p_n \& \text{fin} . C_1 \oplus \text{done} . \dots . C_n \oplus \text{done} . \text{end}$$

Here, all philosophers other than the first must wait for the previous to **finish** eating before they can request to take their chopsticks. The updated protocol $\{p_1 : S_1, p_i : S_2, p_n : S_3, q', C_j, L\}_{j \in 1..n}^{i \in 2..n-1}$ now typechecks for $\varphi = \text{terminating}$.

The previous examples demonstrate how recursion hidden by a universal receive can be used to mimic changes in state. Our final example does the inverse, *i.e.*, we show how a universal receive hidden by a recursive binder can be used to

model resources which eventually reach some permanent state. In addition, we show that universal receives model *fair races*, since they do not impose an order on how communication is handled.

Example 12 (Auction). A merchant m sets up an auction A to accept bids from some buyers b . A merchant can employ different mechanisms for choosing who to sell to (e.g., first come first served, highest bid, biased selling, etc.); but must always respond to buyers with either a *yes*, *no*, or *not-avail* message.

$$\begin{aligned}
 &A : !\alpha \& \text{bid}(\text{int}). m \oplus \text{bid}(\text{int}, \alpha). \text{end} \\
 &m : \mu t. A \& \text{bid}(\text{int}, \beta). \beta \oplus \left\{ \begin{array}{l} \text{yes. } !A \& \text{bid}(\text{int}, \kappa). \kappa \oplus \text{not-avail. end} \\ \text{no. } t \end{array} \right\} \\
 &(b_i)_{i \in 1..n} : \mu t. A \oplus \text{bid}(\text{int}). m \& \{ \text{yes. end, no. } t, \text{ not-avail. end} \}
 \end{aligned}$$

Buyers b_i race to send *bids* to the auction service. In turn, the auction forwards bids and buyer role identifiers to the merchant, who processes bids sequentially (but still in an arbitrary order). If the merchant declines a bid, then the client is offered another chance; if the merchant accepts a bid, then it exposes a replicated receive which informs any further buyers that the product is no longer available. It is key to note that, unlike in Example 11 where we used locks to avoid race conditions, races here are not only allowed but are integral to the protocol. Additionally, by uncovering a replicated receive, the merchant enters a *permanent* state. These two characteristics guarantee that, no matter the selling algorithm employed by the merchant: (i) bids always arrive in a fair arbitrary order; and (ii) the product can only be sold once.

Discussion. We have now shown that *replication* and *recursion* are mutually non-inclusive, and that our extension increases the expressiveness of MPST. It is important to understand the dependencies between added features and the expressiveness gained. Since MPST! is a *conservative* extension, it is guaranteed that the increase of expressiveness derives from our two extensions: 1. the addition of *replication*; and 2. the addition of *first-class roles*.

Replication alone is enough to increase the expressiveness of MPSTs w.r.t. the Chomsky hierarchy. We note that, e.g., Example 8 could be easily re-written without the universal receive, especially since it should not be used by multiple clients to uphold deadlock-freedom—thus, *replication in MPSTs increases their expressiveness to that of context-free languages*.

First-class roles in our formalism refers to: (i) *universal receives* acting as binders on role variables; and (ii) the ability to *pass roles* as payloads in messages. Universal receives allow *protocols to be designed agnostic of the client pool* (consider Examples 2, 6, 9 and 10); and also act as a *fair way of introducing races*—e.g. Example 2 describes a load balancer that responds to requests in *no particular order*. Role passing allows for *safe distributed choice*. In a load balancer (in general), it is *impossible* for a client to know which worker will service its request. In Example 2, role passing allows the server to inform clients of its choice, and also informs the worker of the identity of the client. Role passing

increases expressiveness by introducing dependencies into a protocol. For instance, Example 12 uses role passing to ensure the merchant correctly services the right buyers; without it, a merchant could not respond to requests without bias.

As a final note, first-class roles are different to, *e.g.*, *delegation* or *multiple sessions* (both supported in MPST!) since they act *inside* a session. Therefore our system can still be used to check for properties such as deadlock-freedom, which is not possible with interleaved sessions without other mechanisms such as an interaction typing system [3] or priorities [8].

3.2 Decidability

As one may expect, the added expressiveness of replication and first-class roles into types does not come without a cost. Unlike the base theory that this work extends, even though our language models synchronous communication, typechecking may be *undecidable* in the general case. In the following, we discuss decidability of typechecking in detail. We show that typechecking is only as decidable as the safety property; we provide examples of types that make typechecking problematic; and we provide strategies for determining whether a protocol is captured by a decidable subset of MPST!

Theorem 3 (Decidability of type checking). *If φ is decidable, then type-checking is decidable.*

Proof. Since typing rules in Figure 6 can be deterministically applied based on the structure of a process P , and a typing context need only be split a finite number of times to separate all linear types, there are a finite number of contexts that can be tried for each rule that requires a context split. Lastly, subtyping is decidable [13] (decidability of subtyping replicated types is equivalent to regular branching types, and of parallel types is equivalent to checking multiple session types); and φ is decidable by assumption. $\square$

Theorem 3 states that decidability of type checking is only as decidable as property φ . In Example 14, we will demonstrate why φ may not necessarily be decidable in the general case for the type semantics presented in Figure 7. To do this, we first define *behavioural sets* of type contexts (as in [26, appendix K]).

Definition 8 (Behavioural set). *The behavioural set of a type context, written $\text{beh}(\Gamma)$, is given by $\text{beh}(\Gamma) = \text{unf}^*(\{\Gamma' \mid \Gamma \rightarrow^* \Gamma'\})$; where unf^* is the closure of unf —a function that unfolds all top-level recursive binders in a set of contexts. (Full definitions of unf and unf^* are standard [20, Appendix D].)*

Informally, the behavioural set of a context Γ is the set of (i) its reductions; and (ii) its reductions' unfoldings. The benefit of beh is that it mechanically abides by conditions [S- μ] and [S- $\rightarrow$] from Definition 6. Therefore, to determine whether $\text{beh}(\Gamma)$ is a safety property, all that is required is to exhaustively check the contexts that inhabit $\text{beh}(\Gamma)$ against the remaining conditions of Definition 6.

Example 13. Consider a context $\Gamma = \{s[p] : \mu t. q \oplus m. t, s[q] : \mu t'. p \& m. t'\}$. The behavioural set of Γ is given by:

$$\text{beh}(\Gamma) = \left\{ \left\{ \begin{array}{l} s[p] : \mu t. q \oplus m. t, \\ s[q] : \mu t'. p \& m. t' \end{array} \right\}, \left\{ \begin{array}{l} s[p] : q \oplus m. \mu t. q \oplus m. t, \\ s[q] : p \& m. \mu t'. p \& m. t' \end{array} \right\} \right\}$$

Notice that the left element is the original context after 0 reduction steps, whereas the right element is the unfolding of Γ . Moreover, any further reductions only yield contexts (and unfoldings) already captured by these two elements.

The next example context is problematic for typechecking.

Example 14. Consider a context $\Gamma = \{s[p] : \mu t. q \oplus m. t, s[q] : !p \& m. r \oplus m\}$. The behavioural set of Γ is given by:

$$\text{beh}(\Gamma) = \left\{ \left\{ \begin{array}{l} s[p] : \mu t. q \oplus m. t, \\ s[q] : !p \& m. r \oplus m \end{array} \right\}, \left\{ \begin{array}{l} s[p] : q \oplus m. \mu t. q \oplus m. t, \\ s[q] : !p \& m. r \oplus m \end{array} \right\}, \right. \\ \left. \left\{ \begin{array}{l} s[p] : \mu t. q \oplus m. t, \\ s[q] : !p \& m. r \oplus m \mid r \oplus m \end{array} \right\}, \left\{ \begin{array}{l} s[p] : q \oplus m. \mu t. q \oplus m. t, \\ s[q] : !p \& m. r \oplus m \mid r \oplus m \end{array} \right\}, \dots \right\}$$

Indeed, $\text{beh}(\Gamma)$ is *infinite*. This is a result of how replication and parallel composition are modelled in Figure 7. In fact, the type semantics for replicated communication allows for context reduction to yield *larger* types. Note how in this example, the contexts that inhabit $\text{beh}(\Gamma)$ get infinitely larger by pulling out infinitely many copies of type $r \oplus m$.

Furthermore, we point out that infinite behavioural sets are not only a result of recursive communication with replicated branches. Consider, *e.g.* a $\Gamma' = \{s[p] : !\alpha \& m. \alpha \oplus m'. r \oplus m, s[q] : p \oplus m. !\beta \& m'. \beta \oplus m\}$. Such a context will also pull out infinitely many copies of type $r \oplus m$, because the replicated communication forms an infinite loop. Lastly, it is key to note that $\text{beh}(\Gamma'')$ is finite for any Γ'' that does not contain replicated branches, since there is no other way for a context reduction to yield a larger type.

Knowing whether $\text{beh}(\Gamma)$ is (in-)finite is key for our main decidability result.

Theorem 4 (Decidability of beh). *Let $\varphi = \text{beh}(\Gamma)$. If $\text{beh}(\Gamma)$ is finite, then φ is decidable.*

Proof. Since $\text{beh}(\Gamma)$ contains all possible reducts and unfoldings of Γ , then conditions $[S \rightarrow]$ and $[S \mu]$ are satisfied immediately. Therefore, to determine whether $\text{beh}(\Gamma)$ is a safety property, we may exhaustively check all inhabitants of $\text{beh}(\Gamma)$ against conditions $[S \oplus \&]$, $[S ! \oplus \&]$, $[S \alpha]$, which is decidable since $\text{beh}(\Gamma)$ is finite (by assumption); and since subtyping and frv are decidable. $\square$

Theorem 4 states that φ is *decidable* for any $\varphi = \text{beh}(\Gamma)$ where $\text{beh}(\Gamma)$ is a finite set. In other words, if a protocol can be shown to have a finite behavioural set, then typechecking for that protocol is *decidable*. This could be done manually for each protocol; however, to further increase the practicality of our type system, we present two strategies for restricting protocols into a subset of MPST! with finite behavioural sets.

Decidability Strategies. The strategies we present for restricting protocols to decidable subsets of MPST! all follow a similar blueprint. Essentially, we wish to establish properties on Γ with decidable approximations that imply that $\text{beh}(\Gamma)$ is finite. Then, by Theorems 3 and 4 we obtain decidable typechecking.

The following defines, and gives examples, of each strategy; then, we show these strategies are sound and discuss how they can be approximated.

Definition 9 prevents types like Γ in Example 14 using a naïve approach; put simply, tf captures protocols where all clients of a replicated server are intrinsically non-recursive and non-replicated.

Definition 9 (Trivially finite). A context Γ is trivially finite, $\text{tf}(\Gamma)$, iff:

1. no type in the body of a recursive binder sends to a replicated branch; and
2. continuations of replicated branches do not send to other replicated branches.

Example 15. The protocols modelling the dining philosophers problem in Example 11 are *trivially finite*. Note how the chopstick services make the initial request to the lock service *before* they offer their replicated branch.

For other protocols we need a more nuanced strategy. Definition 10 formalises “loops” in a protocol which may result from replicated servers infinitely bouncing messages amongst each other (such as Γ' in Example 14).

Definition 10 (Loop free). Given a protocol Ψ , and a context Γ derived from Ψ (possibly after a number of reductions), a **cycle** in the LTS of Γ is defined as the series of transitions s.t., for $\Gamma = \Gamma' \cdot s[p] : !\rho \&_{i \in I} m_i(\tilde{T}_i).S_i$

$$\Gamma \xrightarrow{s:q,p:m_k} \left(\frac{s:p'_j, q'_j : m'_j}{j \in 1..n} \right) \xrightarrow{s:q,p:m_k} \Gamma''$$

where $k \in I$, and for any p', q', m', n, Γ'' . A cyclic replicated communication path (**CRCP**) is defined as a cycle with these added conditions:

1. $\Gamma(s[q]) = S_q$ s.t. either $S_q \equiv !S^{\&} | U$ or S_q appears after a recursive binder in $\Psi(q)$, for any $S^{\&}, U$; and
2. $\forall x \in 1..n : \Gamma \xrightarrow{s:q,p:m_k} \left(\frac{s:p'_l, q'_l : m'_l}{l \in 1..x-1} \right) \Gamma''' \xrightarrow{s:p'_x, q'_x : m'_x} \Gamma'''$ we have:
 $\Gamma'''(s[q'_x]) = S_{q'_x}$ s.t. either $S_{q'_x} \equiv !S^{\&} | U$ or $S_{q'_x}$ appears after a recursive binder in $\Psi(q'_x)$.

We say Γ is **loop free**, written $\text{lf}(\Gamma)$, iff the LTS of Γ does not contain a CRCP.

Essentially, a *cycle* in the LTS of a context Γ : (i) starts with an incoming communication action into a replicated type; (ii) performs some intermediary transitions; and (iii) ends with the transition that began the cycle. A CRCP is a special case of a cycle, where all intermediary transitions must also be between roles that have a replicated type; or form part of the body of a recursive type. Finally, a context is *loop free* iff its LTS does not produce any CRCPs.

Example 16. Contexts Γ and Γ' from Example 14 contain CRCPs: Γ contains a CRCP at q with 0 intermediary transitions; and Γ' contains two CRCPs, at q and p , both with 1 intermediary transition forming part of a replicated type.

Example 17. The protocols in Examples 2, 8 and 9 are *loop free*: Example 2 because there are no cycles; Examples 8 and 9 because the cycle between the pong branch on T (resp. M) and the ping branch on P (resp. p_1, p_2) includes communication with the (non-replicated/-recursive) client; breaking the CRCP.

Proposition 1. *If $\text{beh}(\Gamma)$ is infinite, then Γ contains a CRCP.*

Proof. From the type semantics (Figure 7), we observe that the only reductions that can yield *larger* types are communications with replicated branches. Therefore, it follows that for $\text{beh}(\Gamma)$ to be infinite, there must be some reoccurring transitions in the LTS of Γ that repeatedly communicates with a replicated branch—*i.e.*, there is a communication action on a replicated branch, followed by any number of intermediary transitions which then end with the initial communication action on the replicated branch; where all intermediary transitions must be non-finite. This is the definition of a CRCP (Definition 10). $\square$

Theorem 5 (Strategy soundness). *Given a context Γ , $\Phi(\Gamma)$ implies $\text{beh}(\Gamma)$ is finite, for $\Phi \in \{\text{tf}, \text{lf}\}$*

Proof. *Case tf.* By contradiction: assume $\text{beh}(\Gamma)$ is infinite, then, by Proposition 1, Γ contains a CRCP; but, by Definition 10, a CRCP will violate at least one of the conditions for tf in Definition 9—contradiction.

Case lf. By contradiction: assume $\text{beh}(\Gamma)$ is infinite, then, by Proposition 1, Γ contains a CRCP—contradiction; therefore $\text{beh}(\Gamma)$ is *finite*. $\square$

Approximations. Properties tf and lf are both *decidable* for all protocols *without* first-class roles: tf can be determined via a linear traversal of a type context; and lf can be checked by constructing a directed graph of visited replicated branches in a context, then checking that the graph is acyclic (which is decidable). An approximation is only required for protocols using role variables, since their values can only be known at runtime. This approximation would treat any role variable in a selection type to have the capability of reaching *any other role*.

Example 18 (Approximation false negative). Consider the following protocol:

$$p : !\alpha \& m. \alpha \oplus m' \quad q : p \oplus m. p \& m'. r \oplus m'. r \& m \quad r : !\beta \& m'. \beta \oplus m$$

Although the above is *trivially finite* (p and r do not communicate), it would be *falsely* flagged as *not tf* because α is over-approximated to include r .

It is key to note that false negatives of the approximation are avoidable by *requiring unique branching labels on replicated types*. Furthermore, even with the approximation, all examples presented in this paper (except Example 12) are captured by either lf or tf. With respect to Example 12, the presented protocol still yields a finite behavioural set, and thus by Theorem 4 and Theorem 3, typechecking it is decidable. We aim to continue exploring further strategies (especially ones which can capture protocols such as Example 12) in future work.

4 Related Work

The $\text{MAG}\pi$ calculus [19] makes use of generalised MPST theory in order to type *failure-prone* communications (i.e., message loss, reordering, and delays). Key to their approach is the use of timeouts to detect and handle message loss; as with related approaches (e.g., Barwell et al. [2]), this often means that session types are made more complex in order to handle each potential failure point. Our approach is most closely related to that of Le Brun & Dardha [4], who introduce $\text{MAG}\pi!$ as a modification of $\text{MAG}\pi$ to incorporate type-level replication: this has the advantage of simplifying client-server interactions by only requiring *clients* to handle potential failures. However, the aims of our work and that of Le Brun & Dardha [4] are significantly different: while their aim is specifically to use replication as a methodology to simplify failure handling, our work is a more fundamental study of the consequences of type-level replication on expressiveness and decidability. In particular we make use of a more standard base MPST calculus (i.e., a calculus that does not include undirected receives, nor rules that model failures and message reordering), and we make use of *synchronous* communication semantics. Nevertheless our calculus allows for more interesting use of replication: unlike $\text{MAG}\pi!$ we allow *nested replication* and *recursion*, whereas $\text{MAG}\pi!$ only allows replication at the top-level and processes must be finite. Furthermore, as a result of our inclusion of *first-class roles*, as well as using replication *in tandem with* recursion, we can type protocols that make non-trivial use of mutual exclusion and races, all of which would be *inexpressible in $\text{MAG}\pi!$* .

Toninho & Yoshida [30] assess the relative expressiveness of a multiparty session calculus and a process calculus inspired by classical linear logic, showing that MPST calculi allow strictly more expressive process networks (i.e., those that can include cycles). As part of this investigation they explore a limited form of type-level replication that permits a liveness property. However, their system does not consider first-class roles and pre-dates generalised MPST so is guided primarily by global types, and is therefore less expressive than $\text{MPST}!$.

Replicated session types have been used to a limited extent in a wide variety of works on binary session types (e.g., [6,8,5,32]), primarily in works pertaining to Curry-Howard interpretations of propositions as session types, where the exponential modality from linear logic $!A$ is typically linked to replication from the π -calculus. Several further lines of work investigate client-server communication following this correspondence. Kokke et al. [18] investigate an extension of the logically-inspired HCP calculus [17] with two dual modalities $!_n A$ and $?_n A$ to type a pool of n clients and a replicated server that can service n requests respectively, and show that their calculus allows nondeterministic behaviour while still preventing deadlocks and ensuring termination. Qian et al. [23] develop CSLL (client-server linear logic) that uses the dual *coexponential* modalities $\mathfrak{j}A$ and $\mathfrak{i}A$ to type servers and client pools respectively, along with rules to merge client pools. The subtle difference between the $\mathfrak{j}A$ modality and the exponential $!A$ being that the former (informally) serves type A *only as many times as required according to client requests*. This is similar to how our type system operates, given that replicated receives only pull out copies of continuations upon communication.

Multiple requests induce non-determinism into further reductions; in our work this is observed through parallel types, which in the work of Qian et al. [23] is observed through hyperenvironments [17,12]. Unlike all these works, we focus on *multiparty* session types, where interacting with a replicated channel spawns a process that *remains in the same session*. This results in our key novelty, *i.e.*, our account of replication in the *type semantics* with the use of parallel types.

Marshall & Orchard [21] investigate the effects of adding a *semiring graded necessity* modality (a generalisation of the $!$ modality) to a session-typed λ -calculus, showing interesting consequences such as replicated servers and multicast communication. We posit that the two systems can type different protocols: while MPST! cannot straightforwardly encode multicast communication, it is difficult to see how their approach would scale to the examples we describe in Section 3.

Rocha & Caires [24] introduce CLASS, a process calculus with a correspondence to Differential Linear Logic [11]. CLASS integrates session-typed communication, reference cells with mutual exclusion, and replication. Their calculus guarantees preservation and progress, the proof of the latter property requiring a logical relation. CLASS can encode the dining philosophers problem, making essential use of shared state; in contrast our implementation relies on the interplay between replication and recursion.

Deniérou et al. [10] introduce parameterised MPST as a means of designing protocols for parallel algorithms. Their formalism allows for parameterisation of participants in the form of *client*[i], representing the i^{th} client from some group of n clients, for a bound n . The key difference between this formalism and MPST! is that our approach preserves, and allows for the fair handling of, *races*. Parameterised MPST enforce a predetermined prioritisation on the order of communication (thus, Example 12 cannot be expressed in that system).

5 Conclusion

We presented MPST!, a conservative extension of the standard multiparty session π -calculus which introduces for the first time *replication* and *first-class roles*, and proved its metatheory. We have shown that the interplay between replication and recursion allows us to describe interesting and previously inexpressible MPST protocols such as those that rely on races and mutual exclusion, as well as giving a method by which we can express context-free protocols. Although replication can have implications for decidability of typechecking, we have identified sufficient conditions that can determine decidability and provided syntactic approximations for decidability. For future work, it would be interesting to investigate an extension of MPST! with polymorphism, as this would improve on the modular design of protocols already promoted by the type system. Furthermore, we wish to continue exploring the decidability of typechecking to find more general approximations.

Acknowledgements. We thank the anonymous reviewers for their detailed and insightful comments. This work was supported by EPSRC Grants EP/X027309/1 (Uni-pi) and EP/T014628/1 (STARDUST).

References

1. Almeida, B., Mordido, A., Thiemann, P., Vasconcelos, V.T.: Polymorphic lambda calculus with context-free session types. *Inf. Comput.* **289**(Part), 104948 (2022). <https://doi.org/10.1016/j.ic.2022.104948>, <https://doi.org/10.1016/j.ic.2022.104948>
2. Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12–16, 2022, Warsaw, Poland. LIPIcs, vol. 243, pp. 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPIcs.CONCUR.2022.35>, <https://doi.org/10.4230/LIPIcs.CONCUR.2022.35>
3. Bettini, L., Coppo, M., D’Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19–22, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5201, pp. 418–433. Springer (2008). https://doi.org/10.1007/978-3-540-85361-9_33, https://doi.org/10.1007/978-3-540-85361-9_33
4. Le Brun, M.A., Dardha, O.: Mag π !: The role of replication in typing failure-prone communication. In: Castiglioni, V., Francalanza, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 44th IFIP WG 6.1 International Conference, FORTE 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17–21, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14678, pp. 99–117. Springer (2024). https://doi.org/10.1007/978-3-031-62645-6_6, https://doi.org/10.1007/978-3-031-62645-6_6
5. Caires, L., Pérez, J.A.: Linearity, control effects, and behavioral types. In: Yang, H. (ed.) 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. Lecture Notes in Computer Science, vol. 10201, pp. 229–259. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_9, https://doi.org/10.1007/978-3-662-54434-1_9
6. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. *Math. Struct. Comput. Sci.* **26**(3), 367–423 (2016). <https://doi.org/10.1017/S0960129514000218>, <https://doi.org/10.1017/S0960129514000218>
7. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15–19, 2015, Advanced Lectures. Lecture Notes in Computer Science, vol. 9104, pp. 146–178. Springer (2015). https://doi.org/10.1007/978-3-319-18941-3_4, https://doi.org/10.1007/978-3-319-18941-3_4
8. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10803, pp. 91–109. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_5, https://doi.org/10.1007/978-3-319-89366-2_5

9. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. *Inf. Comput.* **256**, 253–286 (2017). <https://doi.org/10.1016/j.ic.2017.06.002>, <https://doi.org/10.1016/j.ic.2017.06.002>
10. Deniérou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. *Log. Methods Comput. Sci.* **8**(4) (2012). [https://doi.org/10.2168/LMCS-8\(4:6\)2012](https://doi.org/10.2168/LMCS-8(4:6)2012), [https://doi.org/10.2168/LMCS-8\(4:6\)2012](https://doi.org/10.2168/LMCS-8(4:6)2012)
11. Ehrhard, T.: An introduction to differential linear logic: proof-nets, models and antiderivatives. *Math. Struct. Comput. Sci.* **28**(7), 995–1060 (2018)
12. Fowler, S., Kokke, W., Dardha, O., Lindley, S., Morris, J.G.: Separating sessions smoothly. *Log. Methods Comput. Sci.* **19**(3) (2023)
13. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. *Acta Informatica* **42**(2-3), 191–225 (2005). <https://doi.org/10.1007/S00236-005-0177-Z>, <https://doi.org/10.1007/s00236-005-0177-z>
14. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) *CONCUR '93*, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23–26, 1993, Proceedings. *Lecture Notes in Computer Science*, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35, https://doi.org/10.1007/3-540-57208-2_35
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) *Programming Languages and Systems - ESOP'98*, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. *Lecture Notes in Computer Science*, vol. 1381, pp. 122–138. Springer (1998). <https://doi.org/10.1007/BFb0053567>, <https://doi.org/10.1007/BFb0053567>
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1), 9:1–9:67 (2016). <https://doi.org/10.1145/2827695>, <https://doi.org/10.1145/2827695>
17. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.* **3**(POPL), 24:1–24:29 (2019)
18. Kokke, W., Morris, J.G., Wadler, P.: Towards races in linear logic. *Log. Methods Comput. Sci.* **16**(4) (2020)
19. Le Brun, M.A., Dardha, O.: $\text{Mag}\pi$: Types for failure-prone communication. In: Wies, T. (ed.) *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings. *Lecture Notes in Computer Science*, vol. 13990, pp. 363–391. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_14, https://doi.org/10.1007/978-3-031-30044-8_14
20. Le Brun, M.A., Fowler, S., Dardha, O.: Multiparty session types with a bang! (2025), <https://arxiv.org/abs/2501.14702>
21. Marshall, D., Orchard, D.: Replicate, reuse, repeat: Capturing non-linear communication via session types and graded modal types. In: Carbone, M., Neykova, R. (eds.) *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022*, Munich, Germany, 3rd April 2022. *EPTCS*, vol. 356, pp. 1–11 (2022). <https://doi.org/10.4204/EPTCS.356.1>, <https://doi.org/10.4204/EPTCS.356.1>
22. Poças, D., Costa, D., Mordido, A., Vasconcelos, V.T.: System $\mathcal{F}_{\omega}^{\ell}$ with context-free session types. In: Wies, T. (ed.) *Programming Languages and Systems - 32nd*

- European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13990, pp. 392–420. Springer (2023). https://doi.org/10.1007/978-3-031-30044-8_15, https://doi.org/10.1007/978-3-031-30044-8_15
23. Qian, Z., Kavvos, G.A., Birkedal, L.: Client-server sessions in linear logic. *Proc. ACM Program. Lang.* **5**(ICFP), 1–31 (2021). <https://doi.org/10.1145/3473567>, <https://doi.org/10.1145/3473567>
 24. Rocha, P., Caires, L.: Safe session-based concurrency with shared linear state. In: ESOP. *Lecture Notes in Computer Science*, vol. 13990, pp. 421–450. Springer (2023)
 25. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming (artifact). *Dagstuhl Artifacts Ser.* **3**(2), 03:1–03:2 (2017). <https://doi.org/10.4230/DARTS.3.2.3>, <https://doi.org/10.4230/DARTS.3.2.3>
 26. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. *Tech. Rep. 6*, Imperial College London (2018), <https://www.doc.ic.ac.uk/research/technicalreports/2018/6>
 27. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* **3**(POPL), 30:1–30:29 (2019). <https://doi.org/10.1145/3290343>, <https://doi.org/10.1145/3290343>
 28. Stutz, F.: Implementability of Asynchronous Communication Protocols - The Power of Choice. Ph.D. thesis, Kaiserslautern University of Technology, Germany (2024), <https://kluedo.ub.rptu.de/frontdoor/index/index/docId/8077>
 29. Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016*. pp. 462–475. ACM (2016). <https://doi.org/10.1145/2951913.2951926>, <https://doi.org/10.1145/2951913.2951926>
 30. Toninho, B., Yoshida, N.: Interconnectability of session-based logical processes. *ACM Trans. Program. Lang. Syst.* **40**(4), 17:1–17:42 (2018)
 31. Vasconcelos, V.T.: Fundamentals of session types. *Inf. Comput.* **217**, 52–70 (2012). <https://doi.org/10.1016/J.IC.2012.05.002>, <https://doi.org/10.1016/j.ic.2012.05.002>
 32. Wadler, P.: Propositions as sessions. *J. Funct. Program.* **24**(2-3), 384–418 (2014). <https://doi.org/10.1017/S095679681400001X>, <https://doi.org/10.1017/S095679681400001X>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Formal Autograding in a Classroom

Dragana Milovančević^(✉) , Mario Bucev, Marcin Wojnarowski,
Samuel Chassot , and Viktor Kunčák 

EPFL, Station 14, CH-1015 Lausanne, Switzerland
{dragana.milovancevic,mario.bucev,marcin.wojnarowski,
samuel.chassot,viktor.kuncak}@epfl.ch

Abstract. We report our experience in enhancing automated grading in an undergraduate programming course using formal verification. In our experiment, we deploy a program verifier to check the equivalence between student submissions and our reference solutions, alongside the existing testing-based grading infrastructure. We were able to use program equivalence to differentiate student submissions according to their high-level program structure, in particular their recursion pattern, even when their input-output behaviour is identical. Consequently, we achieve (1) higher confidence in correctness of idiomatic solutions but also (2) more thorough assessment of solution landscape that reveals solutions beyond those envisioned by instructors.

Keywords: Functional programming · Automated grading · Program equivalence.

1 Introduction

With the growing numbers of students in programming courses, automated grading of programming assignments has become ubiquitous. The goal of practical and rigorous automated grading has motivated the development of a number of tools used in programming courses [60,58,53,37,66,1,26]. These tools commonly employ automated testing to assess the correctness of student submissions [40].

We observed this general trend in the functional programming course at EPFL. Whereas seven years ago the course counted around 200 students, this number has more than doubled since then. To maintain a reasonable workload for the teaching staff, the course heavily relies on testing to automate the assessment of programming assignments throughout the semester and for the final exam. EPFL also developed one of the first Coursera MOOCs in Europe [41], which also uses automated grading.

While highly automated, testing-based grading comes at the cost of accuracy and feedback quality. Researchers have shown that flaws in test suites and the consequential misclassifications of student solutions can have a negative impact on students [67]. Furthermore, testing-based graders fail to provide solution-specific feedback [11,21] on both correct and incorrect solutions. Finally, in large

programming courses, the use of fully automated grading tools can lead to a situation where most students never get to speak with instructors [24], relying solely on the impersonal feedback from automated tools.

Our work aims to provide deeper and more trustworthy automated feedback, while also allowing instructors to detect and analyse solutions that would benefit from further guidance. This approach would enable high-value interactions with students when needed without significantly increasing the instructors’ workload.

As an example, consider the student submissions from Figure 1, computing the `gcd` of two natural numbers. The submission from Figure 1c exhibits the same input-output behaviour as all the other submissions from Figure 1, but is notably less efficient. This submission would thus typically go unnoticed with a testing-based grader, even though it would be interesting to discuss this approach with students. Furthermore, while all the submissions from Figure 1 have identical input-output behaviour, we can identify two different recursion structures (subtraction-based vs. modulo-based Euclid’s algorithm). We would like our grader to classify submissions not only by correctness, but also based on the underlying program structure. Such motivation is also confirmed by the Rainfall study from functional programming education [17], which illustrates the variety in high-level structures of student submissions using manual analysis. Indeed, such classification allows instructors to quickly get an idea of the different approaches followed by students, and to detect those that would benefit from additional feedback.

In this study, we propose the use of formal verification tools for automated grading and analysis of the space of student submissions. We consider functional induction as the strategy of choice for proving program correctness and recursion as the indicator of the underlying program structure. We use the Stainless verifier for Scala [29], which was previously evaluated on equivalence checking benchmarks, including programming assignments in an offline setting, translated from OCaml to Scala [42]. The theory and encouraging results of past programming languages research already suggest that verification tools can be used as grading assistants for program clustering [42,11], but these do not include in-classroom deployment. In this paper, we report on our experiment using the Stainless verifier as a grading assistant in the *live setting* of an undergraduate programming course at EPFL. To encourage other researchers and educators to adopt verification tools in programming classrooms, we document our experience, what did, and what did not work in practice.

The main contributions of this paper are as follows:

- We report on our pilot experiment deploying the Stainless verifier as an automated grader for introductory exercises in our programming course. We describe our deployment process in detail and share our insights.
- We show that formal verification can reveal variations in the underlying program structure of student solutions, such as different recursion schemas, even when solutions have identical input-output behaviour.
- We publish a new data set with over 700 Scala programs, alongside the exercise material [43].

```

def gcdR(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a == b then a
  else if a > b then
    if b == 0 then a
    else gcdR(a - b, b)
  else
    if a == 0 then b
    else gcdR(a, b - a)

```

```

def gcdS(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if b == 0 then
    a
  else if a >= b then
    gcdS(a-b, b)
  else
    gcdS(b, a)

```

(a) Two programs from the cluster of submissions using subtraction-based Euclid's algorithm. Submission `gcdS` is proven equivalent to the reference solution `gcdR`, and therefore correct.

```

def gcdW(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if b == 0 then a else gcdW(b, a%b)

```

```

def gcdX(a: Int, b: Int): Int = b match
  case 0 => a
  case _ => gcdX(b, a % b)

```

```

def gcdY(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  var x = a
  var y = b
  while (y != 0)
    val temp = y
    y = x % y
    x = temp
  x

```

```

def gcdZ(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a < b then
    gcdZ(b, a)
  if b == 0 then a
  else
    val r = a % b
    if r == 0 then b
    else gcdZ(b, r)

```

(b) Four programs from the cluster of submissions using modulo-based Euclid's algorithm, proven equivalent among themselves, and passing all the tests, but not proven equivalent to our reference solution `gcdR`. The instructor can manually inspect one submission from the cluster to provide feedback for the entire cluster (27 submissions).

```

def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  def checkGcd(a: Int, b: Int, testVal: Int): Int =
    require(testVal > 0)
    if a % testVal == 0 && b % testVal == 0 then testVal
    else checkGcd(a, b, testVal - 1)
  (a,b) match
    case (0,x) => x
    case (x,0) => x
    case (x,y) => if x < y then checkGcd(x,y,x) else checkGcd(x,y,y)

```

(c) Singleton cluster and suboptimal implementation. This submission passes the tests, but is not proven equivalent to any other submission. By failing to classify this program, the grader points to this submission that requires manual inspection, allowing the instructor to detect suboptimal implementation.

Fig. 1: Three clusters of student submissions for the `gcd` exercise, illustrating different recursion schemas.


```
def drop(ls: List[Int], n: Int): List[Int] =
  require(n >= 0)
  val helper = ls.foldLeft((Nil, 1))((base, elem) =>
    val list = base._1
    val count = base._2
    val newList = if count % 3 != 0 then list :+ elem else list
    (newList, count + 1)
  )
  helper._1
```

Unfortunately, we found some incorrect functions. Invalid functions:

drop

Counter—example with the following values:

ls = (1, 2, 3, 4, 5, 6, 7, 8, 9, Nil), n = 0

expected (1, 2, 3, 4, 5, 6, 7, 8, 9, Nil) but got (1, 2, 4, 5, 7, 8, Nil)

Fig. 2: An incorrect submission from the `drop` exercise. Stainless detects a concrete counterexample, which we communicate as feedback to the student, and use on the instructor side by adding it to the test suite.

2 Background: Stainless Verifier

One differentiating characteristic of our study is the use of a formal verifier. Our course uses Scala [49], so we adopt the Stainless verifier [36]. Stainless can prove program termination and the absence of runtime errors such as division by zero. Developers can optionally provide program specification using contracts, such as pre- and post-conditions. Stainless supports only a subset of Scala, with best support for a purely functional subset with ML-style polymorphism.¹ This subset largely aligns with the concepts that we present in our course.

Stainless works by first parsing and type checking the input program using the Scala compiler to generate an abstract syntax tree (AST). Stainless then transforms this AST into an equivalent purely functional program, from which it generates verification conditions using a type checking algorithm [29]. It iteratively unfolds recursive functions [64,61] and uses SMT solvers (Z3 [13], cvc5 [2], Princess [55]) to prove or disprove those verification conditions.

Our deployment makes use of a high-level functionality of Stainless to perform program equivalence checks via automated proofs by functional induction [42]. In this mode, rather than writing pre- and post-conditions, users provide specification in the form of a reference program. Stainless then attempts to prove program correctness via automated equivalence checks against the reference program. Our deployment also exploits the ability of Stainless to generate counterexamples for incorrect programs [64], providing feedback valuable to students and instructors (Figure 2).

¹ Stainless also supports imperative programs [28,6,56].

3 The Experiment

In this section, we describe our experiment deploying a Stainless-backed formal autograder in a second-year undergraduate course that teaches software construction through functional programming to around 400 students.

3.1 Teaching Software Construction Using Scala

The goal of our Software Construction course is to teach functional programming and reasoning about programs, along with software engineering concepts and skills. This includes concepts such as subtyping, polymorphism, structural induction, (tail) recursion, as well as soft skills like debugging, reading and writing specifications, or using libraries. During the semester, students work on 12 homework assignments, designed to produce interesting or practical programs, including games as web applications, dynamical system simulations, and file system traversals. In addition to these graded projects, students work on exercise sets, consisting of smaller problems designed to help grasp the course material, such as evaluation of algebraic expressions, manual recursion elimination, and memoisation.

3.2 Experimental Setup

Our experiment took place during the fall semester of 2023. We prepared four autograded exercises in the form of optional individual short programming assignments to be solved and submitted on a computer. Table 1 describes our exercises. We deployed the exercises in the last week of the course, one month before the final exam. This decision was made in part to avoid interfering with other aspects of the course, which was given for the first time in this form. We provided a dedicated section of the course’s forum for questions and discussions about these optional exercises.

Students were invited to participate in the study by submitting their solutions, with an option to permit the public release of those submissions. The code was automatically graded by running tests and formal equivalence checks against our reference solution. We initially deployed one reference solution per exercise. During the experiment, we added another reference solution for the `drop` exercise, as discussed in Section 4. For each submission, students received automated feedback generated by our system. They were permitted to re-submit their solutions any number of times.

At the end of the course, we further examined the submissions using Stainless as a grading assistant on the instructor side. We gathered all the submissions proven correct, together with the reference solutions, and all the submissions that passed the tests, but the equivalence proof timed out against the reference solution. We then run the equivalence checking for each pair of programs in the entire set, to identify and analyse clusters of provably equivalent submissions.

Section 3.3 describes our deployment and Section 3.4 summarizes the results.

Table 1: Four exercises deployed in our experiment. #P: number of submissions per exercise. #F and LOC: average number of functions and lines of code per submission, respectively.

Name	Description	#P	#F	LOC
drop	Drop every n-th element from a list	373	1.8	13
gcd	Find the greatest common divisor	80	1.3	11
prime	Check if an integer is prime	220	3.7	19
infix	Implement infix boolean operators	46	7.8	17

3.3 Setting Up Stainless as a Grader

We next describe how we adapted the Stainless verifier to be used as a scalable on-premise grading service.

Grading Infrastructure. Our students use the Moodle educational platform to obtain and submit graded assignments. An internal Kubernetes service evaluates the submissions by running specified tests and uploading the grade and the feedback to Moodle. Naturally, we integrate our optional autograded exercises as Moodle assignments. This setup yields a simple process for the students, as there is no need to learn and use additional platforms. We use a custom Moodle plugin [7] to automatically run Stainless as a service upon each assignment submission and to report grades and feedback to students. We create a dedicated Docker image for each exercise, packaged with Stainless and Z3 solver. An orchestrating script collects submissions, feeds them into Stainless along with the reference solutions, and produces feedback from the output of Stainless.

Exercise Setup. To illustrate the usability of our approach, we detail the steps needed to prepare each exercise:

1. Write the problem statement as a markdown file.
2. Set up a dedicated Scala project with the source files.
3. Write the reference solution(s).
4. Write unit tests for students to run their solutions locally.
5. Write a configuration file listing the name of the function(s) and the reference solution(s) for the equivalence checking.
6. Compile everything into a Docker image linked to Moodle.

Here, only Step 5 is specific to formal equivalence checks. The other steps were already in place for the existing testing-based grader.

Feedback Generation. For each submission, students receive a numerical grade, a feedback file with comments for each function, and a log file disclosing a detailed output of Stainless. The feedback file either contains a congratulation message, in case the submission is proven correct, or otherwise an explanation of the encountered error(s), such as in our example shown in Figure 2. We report custom feedback for the following errors, allowing students to iteratively improve their code until they solve each problem:

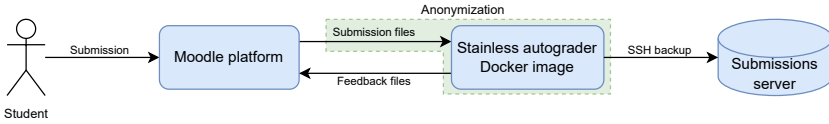


Fig. 3: Data collection pipeline.

- **User errors** – such as an incorrectly named file or an incorrect function signature
- **Safety errors** – such as division by zero or integer overflow
- **Counterexample errors** – a counterexample input was found, proving that the submission is incorrect
- **Termination errors** – a function could not be proven terminating within the specified timeout for SMT queries
- **Equivalence errors** – a function could not be proven equivalent to any reference solution within the specified timeout for SMT queries

Data Collection. Figure 3 shows our data collection pipeline. Upon each submission, the data is directly anonymized and copied out. Each submission is initially identified by a UUID, also included on Moodle to create a one-way link, allowing us to remove collected samples in case a student opts-out of participation.

3.4 Results

Table 1 shows the total number of submissions per exercise, along with the average number of functions per submission and average number of lines of code per submission. Our data set comprising 709 Scala programs is publicly available under a permissive licence [43]. We hope that the data set will be useful to evaluate future research on programming education, which lacks public data sets.²

We export the generated feedback and present the results per exercise in Table 2. Out of the 400 students taking the course, 201 students agreed to participate in the study. Several students actively engaged in discussions on the course forum, with over 70 posts total. While solving the exercises, some students submitted many attempts, and some students skipped some exercises, resulting in a total of 719 submissions. After removing byte-identical files, we were left with 709 submissions. Around one third of the submissions are not of interest due to compilation errors (Column S), resulting in 480 syntactically valid programs.

² In a recent systematic review [40], the authors analyse 121 research papers in the field and remark that, indeed, only 10 of them have publicly available data sets. We are grateful to EPFL’s human research ethics committee and colleagues for their help throughout this study and to our students for allowing us to share their submissions. The overall process involved a significant administrative overhead, which may partly explain the scarcity of public data sets in the literature on programming education.

Table 2: Results of our experiment. S indicates submissions that could not be processed due to compilation errors, or due to students submitting wrong files. TS indicates submissions where Stainless could not prove safety checks. I and C show numbers of submissions proven incorrect and correct, respectively. TO indicates submissions where the equivalence check times out, left for clustering analysis. The last two columns show numbers of non-singleton and singleton clusters.

Name	S	TS	I	C	TO	Non-Singleton	Singleton
drop	60	56	169	14	70	10	26
gcd	21	2	8	3	42	2	12
prime	146	9	40	1	22	4	11
infix	2	0	2	42	0	1	0

Column I shows the number of incorrect submissions, evidenced by a counterexample (46%). Column C shows the number of submissions that our verifier proved equivalent to the reference solution, and therefore correct (13%). The remaining submissions are neither provably correct nor provably incorrect (42%), and are left for further manual inspection and clustering analysis.

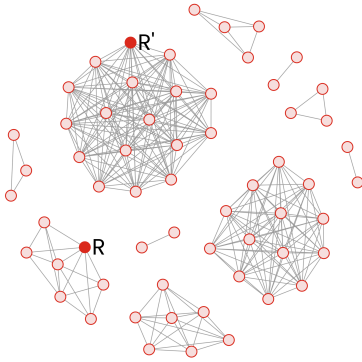
The last two columns show the results of our clustering analysis. Column “Non-Singleton” refers to the number of clusters with two or more equivalent submissions. Column “Singleton” denotes the number of submissions that are not provably equivalent to any other submission, and thus form a singleton cluster of their own.

4 Discussion and Lessons Learned

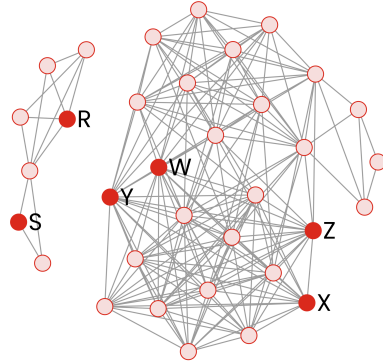
In this section, we interpret the results of our experiment and summarize our takeaways. Figure 4 depicts the clusters of submissions, where nodes represent submissions and edges represent direct equivalence proofs between submissions.

Solution Variations. We find that the verifier’s success rate is lower than in the evaluation in an offline setting [42], with the majority of correct submissions timing out. We believe that this difference is due to the scarcity of reference solutions: in our experiment, we only provided one reference solution per exercise. Based on student feedback, this issue already became apparent during the semester. We addressed this problem by adding another reference solution for the **drop** exercise midway through the experiment, which improved the success rate.

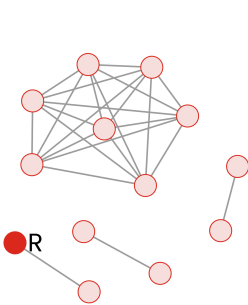
Figure 4a shows the non-singleton clusters of submissions for the **drop** exercise. Our initial reference solution (node R) is in a cluster of size 7. Our additional reference solution (node R’), is in another cluster of size 17. The main difference between the two clusters is in counting to each n -th element, with solutions in the smaller cluster counting backwards from n down to 1 and solutions in the larger cluster counting forward from 1 to n . Another variation of the algorithm counts



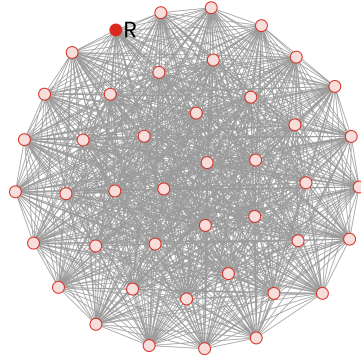
(a) Clusters of submissions for the **drop** exercise. The initial reference solution (node R) is in a cluster of size 7. The second reference solution (node R') is in the largest cluster.



(b) Clusters of submissions for the **gcd** exercise. The reference solution from Figure 1a is in the smaller cluster (node R). Submissions from Figure 1b are in the larger cluster (nodes W, X, Y, Z).



(c) Clusters of submissions for the **prime** exercise. The submissions in the largest cluster compute non-optimized checks for division by each positive integer all the way to the input number.



(d) Clusters of submissions for the **infix** exercise, consisting of 8 non-recursive functions. All the submissions are in the same cluster because there is no recursion to differentiate the structure.

Fig. 4: Non-singleton clusters of submissions originally classified as correct or timed-out for all four exercises. Nodes represent submissions and edges represent direct equivalence proofs. The accompanying artifact contains .gexf files allowing further interactive inspection and graph manipulation via open-source tools such as <https://gephi.org/gephi-lite/>.

forward until the end of the list, computing each time the counter modulo n . This variation is problematic for potential overflows and for termination checks, although in practice we can assume that the size of the input list is smaller than `Int.MaxValue`.

We next consider the non-singleton clusters of submissions for the `gcd` exercise (Figure 4b). Programs from the smaller cluster (7 programs, 2 of which are shown in Figure 1a) use the subtraction-based Euclid’s algorithm. Programs from the larger cluster (27 programs, 4 of which are shown in Figure 1b) use the modulo-based variation. Both clusters correspond to valid solutions for the `gcd` exercise. Furthermore, neither approach can be considered strictly better than the other. **Takeaway:** Even in introductory exercises, a single reference solution is not sufficient to capture the variety in student submissions. One should aim to provide a diverse set of reference solutions to reflect the diversity of student submissions.

Unique Solutions. Some submissions have a unique recursive structure and thus form a cluster of their own (column “Singleton” in Table 2). In the Ask-Elle studies [21], the authors devote particular attention to submissions that do not get matched against any reference solution and belong to a separate “correct (but no match)” category. Singleton clusters also appear in the Rainfall study [17], where they end up in a dedicated “other/unclear composition” category. We inspect the singleton clusters of submissions for the `gcd` exercise and identify the underlying causes for this classification:

- unique solution, identified by its unique recursion schema (Figure 1c)
- limitations of equivalence checking in Stainless (Figure 5a)
- limitations of formal equivalence checking (Figure 5b)

Despite passing all the tests, upon manual inspection, we were able to identify suboptimal implementations among those submissions. In introductory programming courses, such submissions could nevertheless be rewarded the maximum grade, but they are still worthy of custom feedback. On the other hand, in topics such as data science, where performance is of interest, such submissions should preferably be discouraged and only rewarded with partial points [57]. **Takeaway:** By failing to automatically classify certain supposedly correct submissions, the grader can reveal the submissions that require further manual inspection, and help detect suboptimal solutions.

The Role of Instructors. Our grader helped us identify not only unique solutions, but also whole clusters of suboptimal solutions. For example, consider the non-singleton clusters of submissions for the `prime` exercise (Figure 4c). Our reference solution (node R) checks if the input integer n is prime by dividing n by each positive integer up to the square root of n , or until a divisor is found. Interestingly, the most popular solution strategy among students turned out to be checking all the way to n instead.³ Stainless could not prove this strategy

³ This was also the case for the majority of submissions discarded due to compilation errors, implementing the same technique using for-comprehensions.

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  def h(a : Int, b: Int, c: Int = b): Int =
    require(a >= 0 && b >= 0)
    if a < b then gcd(b,a)
    else if c == 0 then a
    else if b == 0 then a
    else if a == b then a
    else if (a%c==0)&&(b%c==0) then c
    else h(a, b, c-1)
  h(a,b)
```

(a) Limitations of Stainless. This program uses the same algorithm as the program in Figure 1c. Yet, the two programs did not end up in the same cluster, due to Stainless attempting at proving the equivalence by decomposing programs into equivalent functions. However, the two inner functions are unfortunately not equivalent, making it impossible for Stainless to conclude the proof.

```
def gcd(a: Int, b: Int): Int =
  require(a >= 0 && b >= 0)
  if a == b then a
  else if a < b then gcd(b, a)
  else if a == 0 then b
  else if b == 0 then a
  else
    val r = a % b
    val q = (a - r) / (a / b)
    gcd(q, r)
```

(b) Limitations of formal equivalence checking. Due to an unusual choice of one student to recompute b as $(a - a \% b) / (a / b)$ on line 9, Stainless was unable to prove the equivalence of this submission to any other submission. Other equivalence checking tools, namely REVE [15] and RVT [23], also resulted in a timeout.

Fig. 5: Singleton clusters from the `gcd` exercise, pointing to submissions that require further manual inspection.

equivalent to our reference solution, forming a separate cluster of size 7. **Takeaway:** Every submission that results in a timeout (passes the tests but is not provably correct) should be checked by instructors. With the help of clustering, the number of manual checks reduces to one representative submission check per cluster. Upon manual inspection, correct submissions can be promoted to reference solutions. This way, the set of tests and reference solutions grows over time, to the point where only new singleton submissions are checked manually.

Library Functions. To tame the disparity between the Scala `List` class and the Stainless `List` class⁴, we provide a stripped version of Stainless `List` with the handout, asking the students to use this version instead of Scala’s. This way, we were able to exploit the benefits of formal verification without discouraging students from using library functions. For example, in the `drop` exercise, we observe submissions using library functions such as `foldLeft`, `length`, and `size`, each forming a separate cluster. One cluster of size two contains tail-recursive programs that use list concatenations. **Takeaway:** Some grading assistants introduce restrictions on using library functions [11]. Yet, support for library functions is crucial,

⁴ The implementation of the `List` class in the Scala library internally mutates the tail for efficiency. To facilitate verification, the Stainless library provides two simpler implementations (an invariant and a covariant list).

as it allows exercising functional programming abstractions. Library functions also lead to more diversity in student submissions.

Rigorous Autograding. Our students receive feedback for each (re)submission, allowing them to make progress from safety errors to logical errors, and finally to error-free programs. This descriptive feedback was well received by the students. Several students reported on the course forum that the grader found bugs in their code that they did not detect locally by running the test suite. For example, in the `gcd` exercise, 7 out of 8 incorrect submissions are due to safety errors in auxiliary functions, that could not be detected by the test suite. The authors of the LAV verifier make similar observations [65]. In their evaluation on C programming assignments, they show that, out of 266 submissions, LAV found 35 incorrect submissions that successfully passed the test suite, mainly due to subtle buffer overflows.

Whereas it is clear that programming assignments are not safety-critical on their own, teaching programming means teaching potential future authors of safety-critical software. Our study advocates for rigorous autograding, in a way that is completely transparent for students (push-button verification [48,62,44]). Namely, students are encouraged to think rigorously about program correctness, without the need for any in-depth knowledge about formal methods. This setting is in line with the vision of formal methods thinking in computer science education put forward by B. Dongol et al. [14] **Takeaway:** Formal verification enriches the feedback given to students. It provides feedback for errors that prevent correctness even if they cannot be formulated easily as counterexamples to program safety.

5 Limitations and Threats to Validity

In this section, we consider some limitations of our approach and our study.

Compilation Errors. We attribute large numbers of compilation errors to students only having local access to the Scala compiler, and not Stainless.⁵ To reduce the number of compilation errors, we should allow students to run Stainless on their local machines. In each problem statement, we should clearly specify which language constructs are supported and which ones are forbidden. Ideally, we should provide local access to Stainless without safety and termination checks. This point is supported by previous research on the role of feedback, which suggests that introducing some form of delay is better for learning purposes, as immediate feedback is prone to undesirable trial-and-error solving strategies [9]. We notice similar concerns in a related report on the Learn-OCaml web platform [27], posing the question whether limiting the number of resubmissions would reduce the number of trivial and syntactic errors.

Syntax and Style. While it would be possible to further incorporate techniques for syntax and style checking [27,30], in this report, we focus on semantic program structure defined by recursion schemas. For example, the infix exercise is

⁵ Stainless only supports a subset of the Scala language (Section 2).

non-recursive, and therefore all the submissions are in the same cluster, despite students using a mixture of if-then-else, pattern matching, built-in boolean and bitwise operators, and custom (non-recursive) boolean functions. Furthermore, program clustering by recursion schemas in some cases does not differentiate purely functional programs from programs with loops and mutations. The reason is that Stainless automatically transforms loops into tail recursive functions during verification [5]. For example, submission `gcdY` with a while loop and variable mutation⁶ is proven equivalent to purely functional submission `gcdW`, and thus placed in the same cluster (Figure 1b).

To address those limitations, future research could consider a more fine-grained clustering to differentiate recursion and loops, by performing syntax checks complementary to inductive equivalence proofs. Similar checks are already used by MOOC graders [27,41].

General Approach. Our analysis only uses data from 709 submissions from one course at one university, and does not necessarily represent the general trend in programming education. We found this setting (200 students, 4 exercises, 15 LOC on average) to be a reasonable size for a pilot study, which enabled us to identify interesting aspects of in-class deployment.

In larger assignments, there is often a clear decomposition into smaller functions, guided by the problem statement or a scaffold. Modular equivalence checkers, such as Stainless, should naturally scale to large assignments that consist of several small functions.

While our study uses the Scala language and the Stainless verifier, our approach applies to other languages and verifiers. In particular, equivalence proofs by functional induction are applicable to recursive programs in general [39] (notable examples include functional induction in Coq [32], recursion induction [47] or computation induction [46] in Isabelle, default induction heuristic in ACL2 [34]). Furthermore, tools such as REVE [15], RVT [23] or SymDiff [35] have shown that equivalence checking is also feasible for imperative programs, beyond the examples that we presented in our study.

6 Related Work

In this section, we describe related work on functional programming education, state-of-the-art clustering-based grading assistants, and applications of formal verification tools in programming courses.

Functional Programming Education. The Rainfall problem [59], originally studied in the field of imperative programming education, has recently become an insightful benchmark in functional programming education. In [17], the author takes over 200 solutions to the Rainfall problem across five functional-first courses and manually splits the solutions based on structure. The Rainfall studies later inspired evaluation of techniques for scaling program classification using

⁶ Stainless supports a limited form of side effects, such as mutation to mostly non-aliased state, and internally handles loops by transformations to recursive functions.

machine learning, assuming that an instructor has indicated categories of interest [12]. In contrast, our approach automatically discovers clusters of submissions using equivalence proofs.

Learn-OCaml is an online grading platform for the OCaml MOOC [8]. Researchers have proposed extensions to the platform allowing assessment of style and test quality [27], as well as understanding how students interact with the grader [20]. Learn-OCaml’s ability to keep track of metrics such as grades, the number of syntax errors and the time spent on each question enables clustering of students into four fixed interaction strategies. In contrast, we consider clustering of submissions, and dynamic discovery of clusters. It would be interesting to combine techniques from Lean-OCaml with our approach and relate interaction strategies to underlying program structures.

Ask-Elle [21] is an online tutor for introductory Haskell exercises, providing feedback and incremental hints using property-based testing and strategy-based tracing. Like in our experiment, the case studies on Ask-Elle observe different patterns in student submissions, and show how Ask-Elle benefits from having multiple reference solutions that provide strategy-specific guidance. Custom feedback in Ask-Elle and Learn-OCaml comes at a cost of significant manual effort to set up a grader. In Ask-Elle, the instructor provides annotations for reference solutions and manually specifies QuickCheck [10] properties. Similarly, for each new Learn-OCaml exercise, the instructor has to specify custom syntax checks, predict unusual solutions, and write mutants to evaluate student-written tests. In contrast, in our deployment, we only had to provide a reference solution for each new exercise, along with optional unit tests, leaving time for instructors to discuss specific solutions with students.

Program Clustering in Grading Assistants. The idea of using equivalence checking to detect algorithmic similarity was previously explored in the ZEUS grading assistant [11]. Like our grader, ZEUS also relies on SMT solving, but while we use functional induction, ZEUS uses inference rules that simulate relationships between expressions. ZEUS is thus more restricted with respect to recursion and introduces limitations for programs with library functions. While both [11] and [42] focus on empirical evaluation of equivalence checking, the focus of our analysis is on *understanding* the resulting clusters and corresponding program structures.

OverCode [22] is a grading assistant for large scale courses, providing an interactive user interface for visualizing clusters of solutions. OverCode supports manual manipulation of program clusters, e.g., merging clusters by adding rewrite rules. However, unlike our approach, OverCode performs neither automated testing nor verification to check for program correctness. Neither OverCode nor ZEUS provide counterexamples for incorrect programs. Complementing our approach with OverCode’s user interface and merge rules could be beneficial to ease manipulation of program clusters and counterexamples.

Verification Tools as Grading Assistants. LEGenT [1] is a tool for personalized feedback generation, using Clara [26] for program clustering and REVE [15] to identify provably correct submissions. LAV [65] is a verification tool evaluated

on imperative programming assignments. Unlike our approach, which supports recursion, both LEGenT and LAV are targeting non-recursive programs.

Dracula [50,63,51] combines the ACL2 theorem prover [34] with the DrScheme graphical interface [16], in introductory programming and software engineering courses. The authors have used Dracula in undergraduate courses on functional programming and software engineering, like is the case in our case study. The main difference is that, in their setting, the goal is to teach students to formulate theorems (programming and proving). In contrast, in our approach, we do not go as far as asking students to systematically prove program properties, or even state them. Instead, in our study, we provide a setting that encourages students to think rigorously about program correctness, without any in-depth knowledge about formal methods. Furthermore, unlike our approach, Dracula does not perform program clustering. Both ACL2 and Stainless have support for automated functional induction, which suggests that it would be possible to perform a similar study in ACL2, even if ACL2 is not higher-order.

Recently, researchers are increasingly sharing their experience on using proof assistants for teaching [3], both for mathematics and computer science programs. Proof assistants have been increasingly finding their way in specialized graduate courses [52,45,33], in undergraduate courses [31,19,54,18,38], and even high schools [25,4]. The question remains whether theorem provers can offer an adequate sufficiently high-level interface for students to write proofs without having to learn the proof assistant itself.

7 Conclusions and Future Work

We have reported our experience in using a formal verifier for evaluation of assignments in an undergraduate programming course. We found that formal verification enriches the feedback given to students. Moreover, verification based on functional induction allowed us to differentiate between solutions, even when solutions exhibit the same input-output behaviour. It allowed us to propose additional reference solutions, and to focus our attention on unusual solutions. We are therefore confident that this approach represents a useful addition to automating assignment evaluation.

We have shared our main takeaways and our deployment process in detail, with the hope that our study will provide inspiration for others trying to incorporate program verifiers and program clustering in assignment assessment in their own courses. In the future, we plan to progressively use our approach on more exercises in our course. To improve the quality of feedback reported to students, we will continuously grow the set of reference solutions and provide more refined verification outcome summaries. In the long run, future research could also consider metrics to explicitly cluster solutions by efficiency. Such feature would be particularly desirable in competitive programming and algorithms courses [57,68].

Availability of Data and Software. The complete data set from this paper is available in an open access Zenodo repository [43].

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Agarwal, N., Karkare, A.: LEGenT: Localizing Errors and Generating Testcases for CS1. In: Proceedings of the Ninth ACM Conference on Learning @ Scale. p. 102–112. L@S’22, Association for Computing Machinery (2022). <https://doi.org/10.1145/3491140.3528282>
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442 (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Bartzia, E., Meyer, A., Narboux, J.: Proof assistants for undergraduate mathematics and computer science education: elements of a priori analysis. In: Trigueros, M. (ed.) INDRUM 2022: Fourth conference of the International Network for Didactic Research in University Mathematics. Reinhard Hochmuth, HAL, Hanovre, Germany (Oct 2022), <https://hal.science/hal-03648357>
4. Bertot, Y., Guilhot, F., Pottier, L.: Visualizing Geometrical Statements with GeoView. Electr. Notes Theor. Comput. Sci. pp. 49–65 (11 2004). <https://doi.org/10.1016/j.entcs.2004.09.013>
5. Blanc, R.W., Kneuss, E., Kuncak, V., Suter, P.: An overview of the Leon verification system: Verification by translation to recursive functions. In: Scala Workshop (2013). <https://doi.org/10.1145/2489837.2489838>
6. Blanc, R.W.: Verification by Reduction to Functional Programs p. 191 (2017). <https://doi.org/https://doi.org/10.5075/epfl-thesis-7636>
7. Bovel, M., Remmal, H.: An automatic grading system for large classes. In: Proceedings of the 52nd Annual Conference of SEFI, Lausanne, Switzerland (2024). <https://doi.org/10.5281/zenodo.14256923>
8. Canou, B., Di Cosmo, R., Henry, G.: Scaling up functional programming education: under the hood of the OCaml MOOC. Proc. ACM Program. Lang. **1**(ICFP) (2017). <https://doi.org/10.1145/3110248>
9. Chevalier, M., Giang, C., El-Hamamsy, L., Bonnet, E., Papaspyros, V., Pellet, J.P., Audrin, C., Romero, M., Baumberger, B., Mondada, F.: The role of feedback and guidance as intervention methods to foster computational thinking in educational robotics learning activities for primary school. Computers & Education **180**, 104431 (2022). <https://doi.org/10.1016/j.compedu.2022.104431>
10. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP ’00 (2000). <https://doi.org/10.1145/351240.351266>
11. Clune, J., Ramamurthy, V., Martins, R., Acar, U.A.: Program Equivalence for Assisted Grading of Functional Programs. Proc. ACM Program. Lang. (OOPSLA) (2020). <https://doi.org/10.1145/3428239>
12. Crichton, W., Sampaio, G.G., Hanrahan, P.: Automating Program Structure Classification. In: Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. p. 1177–1183. SIGCSE ’21 (2021). <https://doi.org/10.1145/3408877.3432358>

13. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08 (2008), https://doi.org/10.1007/978-3-540-78800-3_24
14. Dongol, B., Dubois, C., Hallerstede, S., Hehner, E., Morgan, C., Müller, P., Ribeiro, L., Silva, A., Smith, G., de Vink, E.: On Formal Methods Thinking in Computer Science Education. *Form. Asp. Comput.* (2024). <https://doi.org/10.1145/3670419>, <https://doi.org/10.1145/3670419>, just Accepted
15. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating Regression Verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 349–360. ASE '14 (2014). <https://doi.org/10.1145/2642937.2642987>
16. Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: a programming environment for Scheme. *J. Funct. Program.* p. 159–182 (2002). <https://doi.org/10.1017/S0956796801004208>
17. Fisler, K.: The recurring rainfall problem. In: Proceedings of the Tenth Annual Conference on International Computing Education Research. p. 35–42. ICER'14 (2014). <https://doi.org/10.1145/2632320.2632346>
18. From, A., Jacobsen, F., Villadsen, J.: SeCaV: A sequent calculus verifier in Isabelle/HOL. In: Proceedings of 16th Logical and Semantic Frameworks with Applications. Electronic Proceedings in Theoretical Computer Science, EPTCS, vol. 357, pp. 38–55 (2022). <https://doi.org/10.4204/EPTCS.357.4>
19. Gambhir, S., Guilloud, S., Milovančević, D., Rümmer, P., Kunčák, V.: Lisa tool integration and education plans (2023)
20. Geng, C., Xu, W., Xu, Y., Pientka, B., Si, X.: Identifying Different Student Clusters in Functional Programming Assignments: From Quick Learners to Struggling Students. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education. p. 750–756. SIGCSE 2023 (2023). <https://doi.org/10.1145/3545945.3569882>
21. Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T.: Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *International Journal of Artificial Intelligence in Education* (2016). <https://doi.org/10.1007/s40593-015-0080-x>
22. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.* (2015). <https://doi.org/10.1145/2699751>, <https://doi.org/10.1145/2699751>
23. Godlin, B., Strichman, O.: Regression verification: Proving the equivalence of similar programs. *Software Testing Verification and Reliability* pp. 241–258 (2013). <https://doi.org/10.1002/stvr.1472>
24. Griswold, W.G.: Experience Report: Meet the Professor - A Large-Course Intervention for Increasing Rapport. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. p. 415–421. SIGCSE 2024 (2024). <https://doi.org/10.1145/3626252.3630844>
25. Guilhot, F.: Formalisation en Coq et visualisation d'un cours de géométrie pour le lycée. *Technique et Science Informatiques* pp. 1113–1138 (2005). <https://doi.org/10.3166/tsi.24.1113-1138>
26. Gulwani, S., Radiček, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: Proceedings of the 39th

- ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 465–480. PLDI 2018 (2018). <https://doi.org/10.1145/3192366.3192387>, <https://doi.org/10.1145/3192366.3192387>
27. Hameer, A., Pientka, B.: Teaching the art of functional programming using automated grading (experience report). ICFP (2019). <https://doi.org/10.1145/3341719>
 28. Hamza, J., Felix, S., Kunčák, V., Nussbaumer, I., Schramka, F.: From Verified Scala to STIX File System Embedded Code Using Stainless. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) *NASA Formal Methods*. pp. 393–410. Springer International Publishing, Cham (2022). https://doi.org/https://doi.org/10.1007/978-3-031-06773-0_21
 29. Hamza, J., Voirol, N., Kunčák, V.: System FR: Formalized foundations for the Stainless verifier. OOPSLA, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3360592>
 30. Hart, R., Hays, B., McMillin, C., Rezig, E.K., Rodriguez-Rivera, G., Turkstra, J.A.: Eastwood-Tidy: C Linting for Automated Code Style Assessment in Programming Courses. In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. p. 799–805. SIGCSE 2023 (2023). <https://doi.org/10.1145/3545945.3569817>
 31. Henz, M., Hobor, A.: Teaching Experience: Logic and Formal Methods with Coq. In: Jouannaud, J.P., Shao, Z. (eds.) *Certified Programs and Proofs*. pp. 199–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_16
 32. INRIA: Functional Induction in Coq. <https://coq.inria.fr/refman/using/libraries/funind.html> (2021)
 33. Jacobsen, F., Villadsen, J.: On exams with the isabelle proof assistant. *Electronic Proceedings in Theoretical Computer Science* pp. 63–76 (03 2023). <https://doi.org/10.4204/EPTCS.375.6>
 34. Kaufmann, M., Moore, J.S., Manolios, P.: *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, USA (2000), <https://doi.org/10.1007/978-1-4615-4449-4>
 35. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: *Proceedings of the 24th International Conference on Computer Aided Verification*. p. 712–717. CAV’12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_54
 36. LARA: Stainless. <https://github.com/epfl-lara/stainless> (2023)
 37. Lee, J., Song, D., So, S., Oh, H.: Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. OOPSLA, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3276528>
 38. Maxim, H., Kaliszyk, C., van Raamsdonk, F., Wiedijk, F.: Teaching logic using a state-of-art proof assistant. *Acta Didactica Napocensia* (2010)
 39. McCarthy, J.: A basis for a mathematical theory of computation. In: Braffort, P., Hirschberg, D. (eds.) *Computer Programming and Formal Systems*, *Studies in Logic and the Foundations of Mathematics*, vol. 35, pp. 33–70. Elsevier (1963). [https://doi.org/10.1016/S0049-237X\(08\)72018-4](https://doi.org/10.1016/S0049-237X(08)72018-4)
 40. Messer, M., Brown, N.C.C., Kölling, M., Shi, M.: Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Trans. Comput. Educ.* **24**(1) (2024). <https://doi.org/10.1145/3636515>

41. Miller, H., Haller, P., Rytz, L., Odersky, M.: Functional programming for all! scaling a MOOC for students and professionals alike. In: ICSE Companion. pp. 256–263. ACM (2014)
42. Milovančević, D., Kunčak, V.: Proving and Disproving Equivalence of Functional Programming Assignments. PLDI, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3591258>
43. Milovančević, D., Bucev, M., Wojnarowski, M., Chassot, S., Kunčak, V.: Formal autograding in a classroom (artifact) (2025). <https://doi.org/10.5281/zenodo.14624668>
44. Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 252–269. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3132747.3132748>
45. Nipkow, T.: Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 24–38. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
46. Nipkow, T.: Programming and Proving in Isabelle/HOL. <https://isabelle.in.tum.de/dist/Isabelle2022/doc/prog-prove.pdf> (2022)
47. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002), <https://doi.org/10.1007/3-540-45949-9>
48. Oberhauser, J., Chehab, R.L.d.L., Behrens, D., Fu, M., Paolillo, A., Oberhauser, L., Bhat, K., Wen, Y., Chen, H., Kim, J., Vafeiadis, V.: Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 530–545. ASPLOS '21, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3445814.3446748>
49. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, Fourth Edition (A comprehensive step-by-step guide). Artima, Sunnyvale, CA, USA (2019), https://www.artima.com/shop/programming_in_scala_4ed
50. Page, R.: Engineering software correctness. In: Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education. p. 39–46. FDPE'05 (2005). <https://doi.org/10.1145/1085114.1085123>
51. Page, R., Eastlund, C., Felleisen, M.: Functional programming and theorem proving for undergraduates: a progress report. In: Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education. p. 21–30. FDPE'08 (2008). <https://doi.org/10.1145/1411260.1411264>
52. Pierce, B.C.: Lambda, the ultimate ta: using a proof assistant to teach programming language foundations. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. p. 121–122. ICFP '09 (2009). <https://doi.org/10.1145/1596550.1596552>, <https://doi.org/10.1145/1596550.1596552>
53. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: Sk_p: A neural program corrector for moocs. In: Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. p. 39–40. SPLASH Companion 2016 (2016). <https://doi.org/10.1145/2984043.2989222>
54. Rousselin, P.: Mathematics with Coq for first-year undergraduate students (2023)

55. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 274–289 (2008). https://doi.org/https://doi.org/10.1007/978-3-540-89439-1_20
56. Schmid, G.S., Kuncak, V.: Generalized arrays for stainless frames. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 332–354. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_17
57. Singh, A., Fariha, A., Brooks, C., Soares, G., Henley, A.Z., Tiwari, A., M, C., Choi, H., Gulwani, S.: Investigating student mistakes in introductory data science programming. In: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1. p. 1258–1264. SIGCSE 2024 (2024). <https://doi.org/10.1145/3626252.3630884>
58. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. SIGPLAN Not. p. 15–26 (2013). <https://doi.org/10.1145/2499370.2462195>
59. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. Commun. ACM p. 850–858 (1986). <https://doi.org/10.1145/6592.6594>
60. Song, D., Lee, M., Oh, H.: Automatic and scalable detection of logical errors in functional programming assignments. OOPSLA, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3360614>
61. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) Static Analysis. pp. 298–315 (2011). https://doi.org/10.1007/978-3-642-23702-7_23
62. Tao, R., Shi, Y., Yao, J., Li, X., Javadi-Abhari, A., Cross, A.W., Chong, F.T., Gu, R.: Giallar: push-button verification for the qiskit quantum compiler. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 641–656. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3519939.3523431>
63. Vaillancourt, D., Page, R., Felleisen, M.: ACL2 in DrScheme. In: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications. p. 107–116. ACL2’06 (2006). <https://doi.org/10.1145/1217975.1217999>
64. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala. p. 18–29. SCALA 2015 (2015). <https://doi.org/10.1145/2774975.2774978>
65. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students’ assignments. Inf. Softw. Technol. p. 1004–1016 (2013). <https://doi.org/10.1016/j.infsof.2012.12.005>
66. Wang, K., Singh, R., Su, Z.: Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 481–495. PLDI 2018, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3192366.3192384>
67. Wrenn, J., Krishnamurthi, S., Fisler, K.: Who tests the testers? In: Proceedings of the 2018 ACM Conference on International Computing Education Research. p. 51–59. ICER’18, Association for Computing Machinery (2018). <https://doi.org/10.1145/3230977.3230999>

68. Zhang, J., Li, D., Kolesar, J.C., Shi, H., Piskac, R.: Automated Feedback Generation for Competition-Level Code. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE (2023). <https://doi.org/10.1145/3551349.3560425>, <https://doi.org/10.1145/3551349.3560425>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





COMA, an Intermediate Verification Language with Explicit Abstraction Barriers

Andrei Paskevich^(✉), Paul Patault, and Jean-Christophe Filliâtre

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria,
Laboratoire Méthodes Formelles, 91405 Gif-sur-Yvette, France
andrei.paskevich@universite-paris-saclay.fr

Abstract. We introduce COMA, a formally defined intermediate verification language. Specification annotations in COMA take the form of assertions mixed with the executable program code. A special programming construct representing the abstraction barrier is used to separate, inside a subroutine, the “interface” part of the code, which is verified at every call site, from the “implementation” part, which is verified only once, at the definition site. In comparison with traditional contract-based specification, this offers us an additional degree of freedom, as we can provide separate specification (or none at all) for different execution paths. We define a verification condition generator for COMA and prove its correctness. For programs where specification is given in a traditional way, with abstraction barriers at the function entries and exits, our verification conditions are similar to the ones produced by a classical weakest-precondition calculus. For programs where abstraction barriers are placed in the middle of a function definition, the user-written specification is seamlessly completed with the verification conditions generated for the exposed part of the code. In addition, our procedure can factorize selected subgoals on the fly, which leads to more compact verification conditions. We illustrate the use of COMA on two non-trivial examples, which have been formalized and verified using our implementation: a second-order regular expression engine and a sorting algorithm written in unstructured assembly code.

1 Introduction

Consider a simple function, written in some ML dialect, which eliminates the root node from a binary tree, using an existing library function that merges two trees in one:

```
type tree = Node tree elt tree | Empty

let removeRoot (t: tree) : tree
= match t with
  | Node l _ r → mergeTree l r
  | Empty      → fail
```

If we want to use `removeRoot` in a formally verified program, we need to provide this code with a specification. In a traditional contract-based approach, this means writing a precondition and a postcondition, and here is how they would usually look:

This research was supported, in part, by the ANR project ANR-22-CE48-0013 “GOSPEL” and, in part, by the Décysif project funded by the Île-de-France region and by the French government in the context of “Plan France 2030”.

```

let removeRoot (t: tree) : tree
  requires { t ≠ Empty }
  ensures { match t with
    | Node l _ r → ∀e:elt. e ∈ result ↔ e ∈ l ∨ e ∈ r
    | Empty      → false }

```

While this contract does its job, it is rather unpleasant. Not only does it take more space than the code it describes, it also basically repeats what is already written in the code. What is more, if we compute a verification condition (VC, for brevity) for the definition of `removeRoot`, it will take the form of one `match-with` formula implying another—or maybe two nested `match-with` formulas—and neither is easy to read and to prove.

Some programming languages, like Haskell and Agda, admit multiclause function definitions, and it is tempting to write our specification in this way, too:

```

removeRoot (Node l _ r)
  ensures { ∀e:elt. e ∈ result ↔ e ∈ l ∨ e ∈ r }
= mergeTree l r

removeRoot Empty = fail

```

This definition is much nicer. The postcondition in the first clause can refer to the results of the top-level pattern matching and does not need to do one itself. Furthermore, the second clause is self-explainable, so that we can omit the specification altogether.

However, from the verification point of view, something unusual is happening here. As we push the postcondition down the first branch of the pattern matching, we expose a part of the implementation (namely, the pattern matching itself) to the client code. Whenever `removeRoot` is called in our program, the VC for that call needs to perform the case analysis on the tree parameter in order to access the postcondition. Even more drastically, the second branch contains no specification at all, and so the caller’s VC has to “inline” the entire second clause at the call site and prove that it is never reached.

What we did in this definition of `removeRoot`, is we moved the *abstraction barrier* inwards from the entry-exit boundary of a function, and even omitted it entirely on some of the execution paths. The question is, what are the rules of VC generation for programs with freely moving abstraction barriers? What if we do more in the exposed part of the code than just pattern matching or failure?

In this paper, we propose a formalism for computation and specification that intends to answer this question. We present a programming language called *CoMA* that is small enough to comfortably study its properties, yet expressive enough to serve as a practical intermediate verification language for real-life applications. *CoMA* programs are written in the continuation-passing style—the name *CoMA* is short for Continuation Machine—which allows us to capture with just a few constructions the standard control structures: sequence, conditionals, loops, function calls, exception handling.

Specification annotations in *CoMA* take the form of assertions mixed with executable code. Abstraction barriers are made explicit, as special tags that separate the “interface” part of a subroutine, which is verified at every call site, from the “implementation” part, which is verified only once, at the definition site.

Let us acquaint ourselves with this approach through a few examples. Here, we do not yet use the minimalistic syntax of *CoMA* and stay with familiar ML-like constructs.

First, consider a simple wrapper for the Unix `exit` function, which expects an 8-bit unsigned integer and does not return to the caller. On the left, we show the traditional contract-based specification, and on the right, its rendition in the style of COMA:

<pre>let wrapExit (r: int) requires { 0 ≤ r < 256 } = Unix.exit r</pre>	<pre>let wrapExit (r: int) = assert { 0 ≤ r < 256 }; (↑ Unix.exit r)</pre>

In COMA, the precondition moves into the function body as an assertion which precedes the actual implementation, and we hide the latter under the explicit abstraction barrier denoted $\uparrow$. Since the assertion is put above the barrier, it must be verified at every call of `wrapExit`, whereas the rest of the body is invisible to the caller and produces no proof obligations at call sites. On the definition side, the instructions above the barrier are admitted without verification—which amounts here to simply assuming the asserted property—and the code under the barrier is verified under this assumption. Thus the VC for the definition of `wrapExit` is $\forall r. 0 \leq r < 256 \rightarrow \varphi$, where φ is the VC of the call `Unix.exit r`, whatever it might be.

The representation of postconditions in COMA is more involved, since we need to place them above the abstraction barrier (otherwise, the postcondition would be hidden from the caller), but the actual exit points are in the code below the barrier. We get around this complication by treating the postcondition of a function as the precondition of its continuation. Consider the following function, which, once again, is shown with a traditional specification on the left and rewritten in the COMA style on the right:

<pre>let triple (x: int) : int ensures { result = 3 · x } = x + x + x</pre>	<pre>let triple (x: int) (ret: int → ⊥) = let out (y: int) = assert { y = 3 · x }; (↑ ret y) in (↑ out (x + x + x))</pre>

On the COMA side, the continuation of `triple` is explicit, as a continuation parameter named `ret`. A locally defined function `out` provides a precondition for `ret`, similarly to `wrapExit` above. The definition of `out` is above the barrier in `triple`, and is verified whenever `triple` is called. This produces a VC of the form $\forall y. y = 3 \cdot x \rightarrow \varphi$, where φ is the VC of `ret y`, that is, the verification condition of the rest of the computation, which follows the call of `triple` and is parametrized by the value returned. Note that this is exactly how a classical weakest-precondition calculus would handle calls of the `triple` function on the left-hand side. Conversely, when we verify the implementation of `triple`, the definition of `out` is admitted without proof, and we only need to verify the call of `out` under the barrier. This amounts to proving that `x + x + x` is equal to `3 · x`.

The flexibility and expressive power of explicit abstraction barriers become apparent when we implement the `removeRoot` function in the same fashion:

```
let removeRoot (t: tree) (ret: tree → ⊥) = match t with
| Node l _ r →
  let out (s: tree) = assert { ∀ e: elt. e ∈ s ↔ e ∈ l ∨ e ∈ r };
    (↑ ret s)
  in (↑ out (mergeTree l r))
| Empty → fail
```

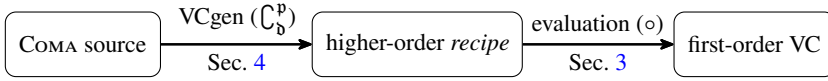


Fig. 1: Verification condition generation in CoMA.

Here, the postcondition and the abstraction barrier are put inside the first branch of the match operation, and the second branch contains no assertions or barriers. This means that the pattern matching and the inadmissibility of `Empty` are part of the interface of `removeRoot`, and will appear in the VC for every individual call of the function.

For non-recursive functions, the barriers can be omitted entirely. In this case, there are no proof obligations generated at the definition site, and the code of the function is effectively inlined during the computation of verification conditions at the call sites.

CoMA is meant to serve as an internal language in program verification tools, rather than as a source language for human programmers. The conversion into continuation-passing style, the introduction of wrapper functions for postconditions, and the placement of barriers should be all performed automatically during translation from the front-end language. For example, when a source function is supplied with a contract, the barriers would protect the entire function body, as for `wrapExit` and `triple` above. A function without a contract would be translated as is, without barriers, and included transparently in the callers' verification conditions. For more complex specifications, like that of `removeRoot`, the front-end language should allow the programmer to express their intent through other means, such as multi-clause definitions, local contracts, etc.

Generation of verification conditions for CoMA is a two-stage process, summarized in Fig. 1. A CoMA program is first translated into a logical proposition, called a *recipe*. Recipes are formulas in a special higher-order logic, with an additional *neutralization operator* which is used for the parts of the code that are admitted without verification. An evaluation procedure, defined as an abstract machine *à la* Krivine, converts recipes into first-order formulas, suitable for automated proving. For CoMA functions specified in a traditional manner, with the barrier at the entry-exit boundary, the resulting proof obligations are similar to those produced by a classical weakest-precondition calculus. However, we can also benefit from the intermediate higher-order form of our verification conditions, and factorize selected subformulas during evaluation. In this way, we curb the well-known exponential growth of classical weakest preconditions, and obtain proof obligations similar to the compact verification conditions of Flanagan and Saxe [7].

We introduce the logic of recipes and the evaluation operator $\circ$ in Sec. 3, together with a number of properties of this logic. The rules of verification condition generation, in the form of a VC operator C_b^p , are given afterwards in Sec. 4. The soundness of our VC generation method is stated in Theorem 3.

We implemented a VC generator for CoMA programs and performed several case studies, two of which are presented here. While in this paper we focus on the pure fragment, our implementation also supports first-class alias-free mutable variables with effect inference and monadic translation into the pure core language. This implementation currently serves as a back-end for CREUSOT, a tool for deductive verification of Rust

programs [5]. Of particular interest for CREUSOT is COMA’s ability to automatically infer the contracts of simple Rust closures (anonymous functions). This and other features are presented in more detail in Section 5.

To summarize, here are the main contributions of our work:

- a new intermediate verification language with higher-order functions and explicit abstraction barriers (Section 2);
- a rigorously defined and proved verification condition generator (Sections 3 and 4);
- a working implementation with numerous added features, including alias-free mutable variables, compact VC formulas via subgoal factorization, pre- and postcondition inference, etc. (Section 5);
- two non-trivial case studies: a second-order regular expression engine (Section 6) and a sorting algorithm written in x86-64 assembly code (Section 7).

An extended version of this paper, containing additional technical definitions and detailed proofs is available online [15].

2 Syntax and Semantics of COMA

The building blocks of COMA are *expressions*, which perform computations, and *terms*, which represent data. Expressions and terms are distinct syntactic entities: a term can be passed as an argument to an expression, but an expression cannot reduce to a term. An expression can be encapsulated in a named or anonymous *handler* (which is what we call subroutines), and either invoked directly or passed as a continuation argument to another expression.

Terms are composed of variables, constants, and pure total operations, provided that they have the same meaning in executable code and in specification. In theory, it would be possible to restrict the syntax of terms to variables and literal values, and delegate all computation to handlers, either predefined or introduced by the user. Still, for the sake of convenience, we admit in terms a handful of basic operations on unbounded integers, Booleans, and polymorphic finite sequences and binary trees. To handle type polymorphism, we treat types as a special kind of data: type expressions are considered to be terms of type *Type*, and we do not make a formal distinction between term and type variables. We denote variables with letters x, y, α, β (the latter two being reserved for types), and terms with s, t, τ, θ (again, the latter two being reserved for types). For specification, we use first-order formulas, denoted φ and ψ , which may contain variables and terms, but not handlers. By a slight abuse of notation, Boolean terms are accepted as atomic formulas.

Handlers accept term parameters (which includes type parameters) and handler parameters, also called continuation parameters or *outcomes*. The list of formal parameters of a handler is called its *type signature*. Since we adopt the continuation-passing style, handlers do not have return values. Handlers that have no parameters are said to have a *void type signature*, written with the symbol $\square$. We use letters π and ϱ to denote type signatures, and letters h, g, f for handler names.

We assume to have access to a number of predefined *primitive handlers*, which form the “standard library” of COMA. Here are the type signatures of five primitive handlers that we use throughout this paper:

type signature:	$\pi, \varrho ::= \overline{x : \tau} \ \overline{g : \varrho}$	parameter list
handler:	$k, o ::= h$ $ \ \pi \rightarrow d$	handler symbol anonymous handler
expression:	$e, d ::= k \ \bar{s} \ \bar{o}$ $ \ e / h \ \pi = d$ $ \ \{ \varphi \} e$ $ \ \uparrow e$ $ \ \downarrow e$	handler call handler definition assertion black-box barrier white-box barrier

Fig. 2: Handlers and expressions.

```

if : (c:bool) (then:□) (else:□)
unTree : (α:Type) (t:tree α) (onNode:(l:tree α) (v:α) (r:tree α)) (onEmpty:□)
get : (α:Type) (s:seq α) (i:int) (return:(v:α))
halt : □
fail : □

```

Handler `if` makes a choice between two continuations, represented by nullary outcomes `then` and `else`, depending on the condition `c`. Handler `unTree` inspects a binary tree `t`: if it is a node, then its datum and two subtrees are passed to the `onNode` continuation, otherwise `onEmpty` is called. Handler `get` retrieves the `i`-th element of sequence `s` and passes it to the continuation. This operation is allowed only when `i` is a valid index of `s`. Handler `halt` stops the computation. Finally, `fail` is an equivalent of `assert false`, it represents code that should never be reached in execution.

By allowing `CoMA` computations to have multiple outcomes, we can represent as first-class entities what usually has to be hardwired into the core syntax of programming languages: conditionals and pattern matching. Handlers `halt` and `fail` are also noteworthy in this regard: as they do not accept continuation parameters, we know simply by looking at their signature that they cannot ever return control to the caller.

Type signatures are identified modulo parameter renaming. For example, the signature of `get` can be equivalently written as $(\beta : \text{Type}) (l : \text{seq } \beta) (k : \text{int}) (ret : (e : \beta))$. Each parameter binds the corresponding symbol in the types of subsequent parameters. This only matters for variables, as handler symbols cannot occur in type annotations.

The *order* of a type signature π (and, by extension, of any handler with that signature) is defined recursively: if π has no continuation parameters, it is of order zero; otherwise, the order of π is one plus the highest order of its outcomes. The length and order of a type signature are invariant with respect to type instantiation: handlers can be polymorphic only in the data types.

Figure 2 presents the syntax of `CoMA` expressions. An expression is an application of a named or anonymous handler to a list of arguments, on top of which we can put recursive *handler definitions*, logical *assertions*, and two *barriers*, denoted $\uparrow$ and $\downarrow$, and called *black-box* and *white-box*, respectively. Handler definitions are placed to the right of the underlying expression; the slash symbol can be read as “where”. The barriers

$$\begin{array}{c}
\frac{h : \pi \in \Gamma}{\Gamma \vdash h : \pi} \text{ (T-SYM)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \square \rightarrow e : \square} \text{ (T-VOID)} \\
\\
\frac{\Gamma \vdash k \bar{s} : (x : \tau) \pi \quad \Gamma \vdash t : \tau}{\Gamma \vdash k \bar{s} t : \pi[x \mapsto t]} \text{ (T-APPT)} \qquad \frac{\Gamma, x : \tau \vdash \pi \rightarrow e : \pi}{\Gamma \vdash (x : \tau) \pi \rightarrow e : (x : \tau) \pi} \text{ (T-PART)} \\
\\
\frac{\Gamma \vdash k \bar{s} \bar{o} : (g : \varrho) \pi \quad \Gamma \vdash k' : \varrho}{\Gamma \vdash k \bar{s} \bar{o} k' : \pi} \text{ (T-APPH)} \qquad \frac{\Gamma, g : \varrho \vdash \pi \rightarrow e : \pi}{\Gamma \vdash (g : \varrho) \pi \rightarrow e : (g : \varrho) \pi} \text{ (T-PARH)} \\
\\
\frac{\Gamma \vdash \varphi : \text{Prop} \quad \Gamma \vdash e : \square}{\Gamma \vdash \{\varphi\} e : \square} \text{ (T-PROP)} \qquad \frac{\Gamma, h : \pi \vdash \pi \rightarrow d : \pi \quad \Gamma, h : \pi \vdash e : \square}{\Gamma \vdash e / h \pi = d : \square} \text{ (T-DEFN)} \\
\\
\frac{\Gamma \vdash e : \square}{\Gamma \vdash \uparrow e : \square} \text{ (T-BBOX)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \downarrow e : \square} \text{ (T-WBOX)}
\end{array}$$

Fig. 3: Typing rules for expressions.

guide the generation of verification conditions, and do not affect execution. The black-box barrier is the abstraction barrier, which separates the exposed “interface” part of a handler definition from the hidden “implementation” part. The white-box barrier is an auxiliary construction that exposes the whole underlying expression. We use letters e and d to denote expressions, and letters k and o to denote handlers.

Type signatures serve as types for expressions, enumerating the expected arguments; in particular, a fully applied handler has type $\square$. *Typing contexts*, denoted Γ and Δ , are sequences of type bindings of the form $x : \tau$ and $g : \varrho$. A typing context is *well-formed* if no symbol is bound twice, and every variable type either is Type or has type Type with respect to the preceding bindings.

The typing rules for expressions are given in Fig. 3. In a judgement $\Gamma \vdash e : \pi$, the typing context is implicitly required to be well-formed. We consider as given the typing relations for terms and formulas, respectively denoted $\Gamma \vdash s : \tau$ and $\Gamma \vdash \varphi : \text{Prop}$; refer to the extended version [15, App. A] for the fragment used in this paper. Notice that bodies of handler definitions and anonymous handlers have to be fully applied. Thus, an anonymous handler $\pi \rightarrow d$ always has type π , modulo parameter renaming. As with type signatures, we identify expressions modulo renaming of bound symbols.

The *initial typing context* Γ_{prim} binds the primitive handlers to their respective type signatures. An expression e is called a *program* when $\Gamma_{\text{prim}} \vdash e : \square$.

We define a small-step operational semantics for COMA as a reduction relation $\longrightarrow$. The reduction rules are shown in Fig. 4. We write $e // \Lambda$ to identify an expression under a series of nested handler definitions, where Λ is a list of definitions, possibly empty. In other words, $e // h_1 \pi_1 = d_1, \dots, h_n \pi_n = d_n$ stands for $e / h_1 \pi_1 = d_1 / \dots / h_n \pi_n = d_n$.

The rule E-SYM expands handler definitions. We assume that no handler is defined in Λ twice, as we can always rename bound handlers. The rules E-APPT and E-APPH perform β -reduction. The rule E-APPC turns an anonymous handler argument into a local handler definition. This is done in a capture-safe manner: we expect that the handler symbol g does not occur freely in d or in $\bar{o}$. The white-box barrier over d is needed to preserve the verification condition of the program, as we show later.

$$\begin{array}{c}
\frac{h \pi = d \in \Lambda}{h \bar{s} \bar{o} // \Lambda \longrightarrow (\pi \rightarrow d) \bar{s} \bar{o} // \Lambda} \quad (\text{E-SYM}) \\
((x : \tau) \pi \rightarrow e) t \bar{s} \bar{o} // \Lambda \longrightarrow (\pi \rightarrow e)[x \mapsto t] \bar{s} \bar{o} // \Lambda \quad (\text{E-APP T}) \\
((g : \varrho) \pi \rightarrow e) f \bar{o} // \Lambda \longrightarrow (\pi \rightarrow e)[g \mapsto f] \bar{o} // \Lambda \quad (\text{E-APP H}) \\
((g : \varrho) \pi \rightarrow e) (\varrho \rightarrow d) \bar{o} // \Lambda \longrightarrow (\pi \rightarrow e) \bar{o} / g \varrho = \downarrow d // \Lambda \quad (\text{E-APP C}) \\
\Box \rightarrow e // \Lambda \longrightarrow e // \Lambda \quad (\text{E-VOID}) \\
\uparrow e // \Lambda \longrightarrow e // \Lambda \quad (\text{E-BBOX}) \\
\downarrow e // \Lambda \longrightarrow e // \Lambda \quad (\text{E-WBOX}) \\
\frac{\vdash \varphi}{\{\varphi\} e // \Lambda \longrightarrow e // \Lambda} \quad (\text{E-PROP}) \\
\frac{h \text{ is not free in } e}{e / h \pi = d // \Lambda \longrightarrow e // \Lambda} \quad (\text{E-GC})
\end{array}$$

$$\begin{array}{c}
\frac{\vdash s}{\text{if } s \text{ } k \text{ } o // \Lambda \longrightarrow k // \Lambda} \quad \frac{\vdash \neg s}{\text{if } s \text{ } k \text{ } o // \Lambda \longrightarrow o // \Lambda} \\
\frac{\vdash t = \text{Node } s_1 \ s_2 \ s_3}{\text{unTree } \tau \text{ } t \text{ } k \text{ } o // \Lambda \longrightarrow k \ s_1 \ s_2 \ s_3 // \Lambda} \quad \frac{\vdash t = \text{Empty}}{\text{unTree } \tau \text{ } t \text{ } k \text{ } o // \Lambda \longrightarrow o // \Lambda} \\
\frac{\vdash 0 \leq s_2 < \text{length } s_1 \quad \vdash s_1[s_2..s_2+1] = [t]}{\text{get } \tau \ s_1 \ s_2 \ k // \Lambda \longrightarrow k \ t // \Lambda}
\end{array}$$

Fig. 4: Operational semantics.

The rule E-PROP requires the asserted formula φ to be valid before proceeding with the execution. The validity judgement $\vdash \varphi$ is made within the standard model for our data types: integers, Booleans, sequences and binary trees. Of course, the validity of an arbitrary proposition cannot be effectively verified in a practical implementation. However, our purpose here is different: we define the operational semantics of COMA in order to state and prove the correctness of our verification procedure—in particular, that a program with a valid verification condition cannot get stuck during its execution because of a failed assertion.

The rule E-VOID replaces a nullary anonymous handler by its body. The barriers are ignored during execution (rules E-BBOX and E-WBOX). Finally, the rule E-GC prunes the context by removing unreachable handler definitions. This rule commutes with the rest of the rules, making COMA non-deterministic, yet still strongly confluent.¹ We can define the operational semantics of COMA without the E-GC rule; however, it helps keeping our definition context clean by removing local handler definitions once the parent handler finishes its computation.

We also postulate the evaluation rules for the primitive handlers. As for E-PROP, the application of these rules depends on validity of logical properties that express the pre- and postcondition of the primitives. For example, the Boolean condition of an `if` must be valid for the evaluation to progress along the first outcome (as mentioned in the beginning of the section, we admit Boolean terms as atomic formulas). In the rules for `unTree`, the function symbols `Node` and `Empty` are the constructors of the `tree` type. In

¹ Modulo semantic equality of answer terms during evaluation of primitive handlers; see below.

```

product (a b: int) (return (c: int))
= { b ≥ 0 }
  (↑ loop a b 0
   / loop (p q r: int)
     = { p · q + r = a · b ∧ q ≥ 0 }
       (↑ if (q > 0) (→ if (q mod 2 = 1) (→ next (r + p)) (→ next r)
                    / next (s: int) = loop (p + p) (q div 2) s)
        (→ break r)))
  / break (c: int) = { c = a · b } (↑ return c)

```

Fig. 5: Russian Peasant Multiplication in COMA.

the rule for `get`, we use the (total) slice operator on sequence s_1 to isolate the element at the position s_2 , which must be a valid index in s_1 . The answer terms in the rules for `unTree` and `get` can be any ground terms that validate the rule premises: for example, the expression `get int [42] 0 return // Λ` can reduce both to `return 42 // Λ` and `return 6*7 // Λ`. While we could introduce some form of normalization to avoid this syntactical divergence, there is no need for that, since all conditions in our evaluation rules are expressed in terms of semantic validity. Finally, `halt` and `fail` represent the final states of a computation and cannot be evaluated.

COMA is a type-safe language. Type preservation is easy to establish, either through a direct proof or by embedding in a more expressive framework like System F_ω or CoC. As for the progress property, the blocking semantics of assertions limits it to programs with a valid verification condition; and so we defer this subject until Section 4.

We conclude with two examples. First, we rewrite `removeRoot` in proper COMA:

```

removeRoot (t: tree elt) (return (s: tree elt)) =
  unTree elt t ((l: tree elt) (_: elt) (r: tree elt) →
    (↑ mergeTree elt l r out)
    / out (s: tree elt) = { ∀ e:elt. e ∈ s ↔ e ∈ l ∨ e ∈ r }
    (↑ return s))
  fail

```

Just like in Sec. 1, the implementation starts with a case analysis on the tree parameter t , using the primitive `unTree` handler. The two branches of the case analysis are represented, respectively, by an anonymous handler, which is called when t is a binary node, and the `fail` primitive, invoked when t is `Empty`. The anonymous handler makes a call of `mergeTree`, which we assume to be available and polymorphic akin to `unTree`. The result of `mergeTree` is passed to a wrapper handler `out` which asserts the postcondition of the first branch, before calling the continuation parameter of `removeRoot`.

In Fig. 5, we show the Russian Peasant Multiplication algorithm written in COMA. This code is specified in a more traditional manner: the entire implementation of the `product` handler is put behind the abstraction barrier. Left in the interface part are the starting assertion $\{b \geq 0\}$, which naturally becomes the precondition of `product`, and the wrapper handler `break`, which plays the same role as `out` in `removeRoot`, and whose precondition $\{c = a \cdot b\}$ is the postcondition of `product`.

The implementation defines and calls a recursive handler named `loop`. This handler does not return to the caller: to do that, it would need to receive a continuation parameter and call it, like `removeRoot` and `product` do. Instead, `loop` escapes by calling `break` at the end of computation. In this respect, `loop` behaves indeed rather like a loop than a recursive function: its continuation is determined statically, by its lexical context, rather than dynamically by its caller. Consequently, there is no distinct postcondition associated to `loop`: in COMA, postconditions are preconditions of continuation parameters (attached via wrapper handlers like `out` and `break`), and `loop` has none thereof. And the precondition of `loop`, placed above the barrier, is just the loop invariant.

In practice, the majority of COMA programs would be generated by mechanical translation from existing languages like OCaml or Rust. Part of this translation would be a CPS transformation, required by our language. While in most cases, this transformation is not problematic, and allows us to reduce a large number of control structures to just two—definitions and calls—there are limits to what can be easily translated into COMA. Consider, for example, an OCaml exception that carries a closure:

```
exception E of (int → int)
```

In COMA, exception-raising functions are written as handlers that have multiple continuation parameters: one for the normal outcome, and one for each exception that might be raised in the handler code. However, if the closures passed with the exception `E` were themselves liable to raise `E`, we would not be able to give them a finite type in COMA. Incidentally, it is not a coincidence that higher-order exceptions can be used to realize fixed point computations without explicit recursion.

3 The Logic of Recipes

In their final form, verification conditions for COMA programs are first-order logical formulas, which we can handle with the usual methods of automated and interactive theorem proving. Their generation, however, goes through an intermediate stage, where a preliminary higher-order verification condition, called *recipe*, is constructed and then transformed, deterministically and in a finite number of steps, into the first-order form.

Recipes are formulas in a particular variety of higher-order logic, where bound predicate variables represent verification conditions of individual handlers and can only appear in a positive position. We denote recipes with letters Φ, Ψ, Υ . The syntax of recipes is given in Fig. 6. In recipes, handler symbols become predicate variables of the same name and arity as the original handler. The symbol $\mathbf{0}$ is the verification condition of `fail`, a logical contradiction. The *neutralization operator*, denoted $\mathfrak{h}$, suppresses proof obligations in the underlying recipe. Finally, notice that the antecedent in an implication is not a recipe, but a first-order formula, which cannot have occurrences of handler symbols. We write $\lambda\pi.\Phi$ and $\forall\pi.\Phi$ to denote a series of nested λ -abstractions or quantifications. By convention, $\lambda\Box.\Phi$ and $\forall\Box.\Phi$ are the same as Φ .

Universal quantification over a predicate variable is defined recursively, as an instantiation with a *joker recipe* $\mathbf{0}_\pi$. A joker recipe is the verification condition of a handler of which nothing is known: on any input, the handler may fail or it may call any of its outcomes with arbitrary arguments. On a void type signature, the joker $\mathbf{0}_\square$ is simply $\mathbf{0}$.

$$\begin{aligned}
\Phi, \Psi, \Upsilon &::= h \mid \Phi \ s \mid \lambda x : \tau . \Phi \mid \forall x : \tau . \Phi \mid \Phi \wedge \Psi \\
&\mid \mathbf{0} \mid \Phi \ \Psi \mid \lambda g : \varrho . \Phi \mid \varphi \rightarrow \Phi \mid \natural \Phi \\
\forall h : \pi . \Phi &\triangleq (\lambda h : \pi . \Phi) \mathbf{0}_\pi \\
\mathbf{0}_\pi &\triangleq \lambda \pi . \mathbf{0} \wedge \bigwedge_{(f : \bar{x} : \bar{\tau} \ \bar{g} : \bar{\varrho}) \in \pi} \forall \bar{x} : \bar{\tau} . \forall \bar{g} : \bar{\varrho} . f \ \bar{x} \ \bar{g}
\end{aligned}$$

Fig. 6: Preliminary verification conditions (recipes).

A fully applied recipe is a logical proposition, which is why we disregard the result type (that is, `Prop`) and use type signatures once again as types for predicate variables and recipes. The typing rules are given in the extended version [15, Fig. 7]. Unlike implication and universal quantification, the conjunction connective applies to any two recipes of the same type, and not only to fully applied recipes. The neutralization operator applies to recipes of any type. We identify recipes modulo renaming of bound symbols.

The semantics of recipes is given by means of a Krivine-style abstract machine [10] that converts a fully applied recipe into a first-order formula, where all bound predicate variables are eliminated. We have chosen this approach both for theoretical and practical reasons. First, the properties of recipes are naturally proved using logical relations [16], which are straightforward to express in this setting. Second, our implementation of a verification condition generator for COMA is based on the same abstract machine. In the rest of the section, we introduce this evaluator and establish some of its properties.

A *cell* is a triplet $\langle b, \Sigma, \Phi \rangle$, where Φ is a recipe, Σ a *cell context*, binding every free handler symbol in Φ to a cell, and b a Boolean value. Such a cell can be converted into a recipe by replacing the free handler symbols in Φ with the corresponding converted cells from Σ . We assign the type signature of the resulting recipe to the initial cell. In what follows, we denote cells with letters C and D , and assume that all cells and recipes under consideration are well-typed.

We associate a specification recipe to each primitive handler:

$$\begin{aligned}
\Psi_{\text{if}} &\triangleq \lambda c : \text{bool} . \lambda \text{then} : \square . \lambda \text{else} : \square . (c \rightarrow \text{then}) \wedge (\neg c \rightarrow \text{else}) \\
\Psi_{\text{unTree}} &\triangleq \lambda \alpha : \text{Type} . \lambda t : \text{tree } \alpha . \lambda \text{onNode} : (l : \text{tree } \alpha) (v : \alpha) (r : \text{tree } \alpha) . \lambda \text{onEmpty} : \square . \\
&\quad (\forall l : \text{tree } \alpha . \forall v : \alpha . \forall r : \text{tree } \alpha . t = \text{Node } l \ v \ r \rightarrow \text{onNode } l \ v \ r) \wedge \\
&\quad (t = \text{Empty} \rightarrow \text{onEmpty}) \\
\Psi_{\text{get}} &\triangleq \lambda \alpha : \text{Type} . \lambda s : \text{seq } \alpha . \lambda i : \text{int} . \lambda \text{return} : (v : \alpha) . \\
&\quad 0 \leq i < \text{length } s \wedge \forall v : \alpha . s[i..i+1] = [v] \rightarrow \text{return } v \\
\Psi_{\text{halt}} &\triangleq \natural \mathbf{0} \qquad \Sigma_{\text{prim}} \triangleq [\text{if} \mapsto \langle \perp, \emptyset, \Psi_{\text{if}} \rangle, \\
\Psi_{\text{fail}} &\triangleq \mathbf{0} \qquad \text{unTree} \mapsto \langle \perp, \emptyset, \Psi_{\text{unTree}} \rangle, \text{get} \mapsto \langle \perp, \emptyset, \Psi_{\text{get}} \rangle, \\
&\quad \text{halt} \mapsto \langle \perp, \emptyset, \Psi_{\text{halt}} \rangle, \text{fail} \mapsto \langle \perp, \emptyset, \Psi_{\text{fail}} \rangle]
\end{aligned}$$

The *initial cell context* Σ_{prim} binds primitive handlers to their respective specifications.

The *depth* of a cell $C = \langle b, \Sigma, \Phi \rangle$ is zero if its cell context Σ is empty; otherwise, it is one plus the maximum depth of the cells in Σ . The *neutralization* of C , denoted $\natural C$, is the cell $\langle \top, \natural \Sigma, \Phi \rangle$, where $\natural \Sigma$ is obtained by neutralizing every cell in Σ . Obviously, neutralization does not affect the type or depth of a cell.

$$\begin{array}{ll}
\langle b, \Sigma, \mathbf{0} \rangle \circ \varepsilon \triangleq b & \langle b, \Sigma, \Phi \wedge \Psi \rangle \circ \ell \triangleq \langle b, \Sigma, \Phi \rangle \circ \ell \wedge \langle b, \Sigma, \Psi \rangle \circ \ell \\
\langle b, \Sigma, h \rangle \circ \ell \triangleq \Sigma(h) \circ \ell & \langle b, \Sigma, \varphi \rightarrow \Phi \rangle \circ \varepsilon \triangleq \varphi \rightarrow \langle b, \Sigma, \Phi \rangle \circ \varepsilon \\
\langle b, \Sigma, \mathfrak{h}\Phi \rangle \circ \ell \triangleq \langle \top, \mathfrak{h}\Sigma, \Phi \rangle \circ \ell & \langle b, \Sigma, \forall x:\tau. \Phi \rangle \circ \varepsilon \triangleq \forall x:\tau. \langle b, \Sigma, \Phi \rangle \circ \varepsilon \\
\langle b, \Sigma, \Phi s \rangle \circ \ell \triangleq \langle b, \Sigma, \Phi \rangle \circ s, \ell & \langle b, \Sigma, \lambda x:\tau. \Phi \rangle \circ s, \ell \triangleq \langle b, \Sigma, \Phi[x \mapsto s] \rangle \circ \ell \\
\langle b, \Sigma, \Phi \Psi \rangle \circ \ell \triangleq \langle b, \Sigma, \Phi \rangle \circ \langle b, \Sigma, \Psi \rangle, \ell & \langle b, \Sigma, \lambda h:\pi. \Phi \rangle \circ C, \ell \triangleq \langle b, \Sigma \uplus [h \mapsto C], \Phi \rangle \circ \ell
\end{array}$$

Fig. 7: Recipe evaluation.

A *stack* is a mixed sequence of terms and cells. An empty stack is denoted ε . The neutralization of a stack ℓ , denoted $\mathfrak{h}\ell$, is obtained by neutralizing every cell in ℓ . We say that a stack is *aligned* with a cell, when the length of the stack and the types of its elements coincide with the cell's signature. In other words, an aligned stack contains appropriate arguments for the cell.

An n -ary relation R on same-typed cells holds on stacks $\ell_1, \dots, \ell_n$ when they all have the same length and type signature, and for each position i , if $\ell_{1i}, \dots, \ell_{ni}$ are terms, then they are all identical, and if they are cells, then both $R(\ell_{1i}, \dots, \ell_{ni})$ and $R(\mathfrak{h}\ell_{1i}, \dots, \mathfrak{h}\ell_{ni})$ are true. Similarly, R holds on cell contexts $\Sigma_1, \dots, \Sigma_n$ when they bind the same handler names, and for every bound h , both $R(\Sigma_1(h), \dots, \Sigma_n(h))$ and $R(\mathfrak{h}\Sigma_1(h), \dots, \mathfrak{h}\Sigma_n(h))$ are true. It is easy to see that $R(\Sigma_1, \dots, \Sigma_n)$ implies $R(\mathfrak{h}\Sigma_1, \dots, \mathfrak{h}\Sigma_n)$, as $\mathfrak{h}\mathfrak{h}C = \mathfrak{h}C$.

As a special case of the above, any property of cells is said to hold for a stack ℓ or a cell context Σ whenever for every cell C in ℓ or Σ , both C and $\mathfrak{h}C$ have this property. Furthermore, if the property holds for a cell context Σ , then it also holds for $\mathfrak{h}\Sigma$.

The evaluation operator $\circ$, defined in Fig. 7, applies a cell to an aligned stack and produces a first-order logical formula. In the rule for $\forall x:\tau. \Phi$, we assume that x does not occur in the cell context Σ , to avoid collisions.

Theorem 1. *The evaluation operator $\circ$ is defined on all aligned cells and stacks.*

Proof. We say that a cell C is *normalizing* if for any aligned normalizing stack ℓ , the evaluation $C \circ \ell$ is defined. This definition is well-founded, because every cell in ℓ is of lower order than C . We need to show that all cells (ergo, all stacks) are normalizing. In fact, it suffices to prove that every cell $\langle b, \Sigma, Y \rangle$ is normalizing, if its cell context Σ is normalizing. Afterward, a simple induction over cell depth allows us to conclude.

We proceed by induction over the size of Y , counting only the subrecipes, so that term substitutions do not affect the size. Take an arbitrary aligned normalizing stack ℓ .

Case Y is $\mathbf{0}$. As $\mathbf{0}$ is $\square$ -typed, ℓ has to be empty, and $\langle b, \Sigma, \mathbf{0} \rangle \circ \varepsilon$ is defined.

Case Y is h . Since every cell in Σ is normalizing, the evaluation $\Sigma(h) \circ \ell$ is defined.

Case Y is $\mathfrak{h}\Phi$. The context $\mathfrak{h}\Sigma$ is normalizing, and the induction hypothesis applies.

Case Y is $\Phi \Psi$. Let D be $\langle b, \Sigma, \Psi \rangle$. Since $\mathfrak{h}\Sigma$ is normalizing, both cells, D and $\mathfrak{h}D$, are normalizing by the induction hypothesis. As the cell $\langle b, \Sigma, \Phi \rangle$ is also normalizing by the induction hypothesis, the evaluation $\langle b, \Sigma, \Phi \rangle \circ D, \ell$ is defined.

Case Y is $\lambda h:\pi. \Phi$. Then the stack ℓ is of the form D, ℓ' , where both D and $\mathfrak{h}D$ are normalizing. Thus, $\Sigma \uplus [h \mapsto D]$ is normalizing and the induction hypothesis applies.

In every other case, we pick a rule for $\circ$ and apply the induction hypothesis. $\square$

A cell C is said to be *neutral*, if for any aligned neutral stack ℓ , the formula $C \circ \ell$ is valid. Just as above, this recursive definition is well-founded, because every cell in ℓ is of lower order than C .

Lemma 1. *Any neutralized cell $\mathfrak{h}C$ is neutral.*

The proof of this and following lemmas can be found in the extended version [15, Sec. 3].

A cell C_1 *entails* C_2 , denoted $C_1 \Rightarrow C_2$, when they have the same type and for any aligned stacks ℓ_1 and ℓ_2 such that $\ell_1 \Rightarrow \ell_2$, we have $C_1 \circ \ell_1 \Rightarrow C_2 \circ \ell_2$. Here and below, the symbol $\Rightarrow$ stands for logical consequence, and $\Leftrightarrow$ for logical equivalence, under the same standard model used for assertions. Cell C_1 is *equivalent* to C_2 , denoted $C_1 \equiv C_2$, when $C_1 \Rightarrow C_2$ and $C_2 \Rightarrow C_1$.

Lemma 2. *Cell entailment is reflexive and transitive.*

Lemma 3. *Consider two cells of the same type, C_1 and C_2 , such that for any aligned stack ℓ , we have $C_1 \circ \ell \Rightarrow C_2 \circ \ell$. Then $C_1 \Rightarrow C_2$.*

Note that $C_1 \Rightarrow C_2$ does not imply $\mathfrak{h}C_1 \Rightarrow \mathfrak{h}C_2$. For example, the cell $\langle \perp, \emptyset, \lambda g : \square. \mathbf{0} \rangle$ entails $\langle \perp, \emptyset, \lambda g : \square. g \rangle$, yet, when we apply their neutralizations to $\langle \perp, \emptyset, \mathbf{0} \rangle$, we obtain $\top$ and $\perp$, respectively. Thus, to establish $\ell_1 \Rightarrow \ell_2$, we must show pairwise entailment not only for the cells in the two stacks, but also for their neutralizations.

Given three cells $C_1 = \langle b_1, \Sigma_1, \Phi \rangle$, $C_2 = \langle b_2, \Sigma_2, \Phi \rangle$, and $C_3 = \langle b_3, \Sigma_3, \Phi \rangle$ that have the same type and the same recipe Φ , we say that C_1 is the *fusion* of C_2 and C_3 when $b_1 = b_2 \wedge b_3$ and Σ_1 is the fusion of Σ_2 and Σ_3 . Quite obviously, if C_1 is the fusion of C_2 and C_3 , then $\mathfrak{h}C_1$ is the fusion of $\mathfrak{h}C_2$ and $\mathfrak{h}C_3$ (all three are actually the same). Furthermore, any cell C is the fusion of itself and $\mathfrak{h}C$.

A cell C_1 is a *meet* of C_2 and C_3 if they all have the same type, the neutralized cells $\mathfrak{h}C_2$ and $\mathfrak{h}C_3$ are equivalent, and for any aligned stacks ℓ_1, ℓ_2, ℓ_3 such that ℓ_1 is a meet of ℓ_2 and ℓ_3 , we have $C_1 \circ \ell_1 \Leftrightarrow C_2 \circ \ell_2 \wedge C_3 \circ \ell_3$.

Lemma 4. *If C_1 is the fusion of C_2 and C_3 , then C_1 is a meet of C_2 and C_3 .*

Corollary 1. *For any cell C and aligned stack ℓ , we have $C \circ \ell \Leftrightarrow \mathfrak{h}C \circ \ell \wedge C \circ \mathfrak{h}\ell$.*

Corollary 2. *Any cell C entails $\mathfrak{h}C$.*

Lemma 5. *Consider cells C_1, C_2, C_3 of the same type, such that $\mathfrak{h}C_2 \equiv \mathfrak{h}C_3$ and for any aligned stack ℓ , we have $C_1 \circ \ell \Leftrightarrow C_2 \circ \ell \wedge C_3 \circ \ell$. Then C_1 is a meet of C_2 and C_3 .*

This leads to a surprising distributivity property. Consider a cell $D = \langle b, \Sigma, \Phi \wedge \Psi \rangle$ and its conjuncts $D_1 = \langle b, \Sigma, \Phi \rangle$ and $D_2 = \langle b, \Sigma, \Psi \rangle$. If $\mathfrak{h}D_1$ is equivalent to $\mathfrak{h}D_2$, then, by Lemma 5, D is a meet of D_1 and D_2 , and $\mathfrak{h}D$ is a meet of $\mathfrak{h}D_1$ and $\mathfrak{h}D_2$. Then, for any appropriate cell C and stack ℓ , the formula $C \circ D, \ell$ is logically equivalent to the conjunction of $C \circ D_1, \ell$ and $C \circ D_2, \ell$. Informally speaking, we can split a recipe over any cell conjunction, no matter where it occurs inside the recipe, as long as the conjuncts have equivalent neutralizations.

Theorem 2. *Consider a type signature π and a cell $J = \langle \perp, \Sigma, \mathbf{0}_\pi \rangle$. For every cell C of type π , we have $J \Rightarrow C$ and $\mathfrak{h}J \Rightarrow \mathfrak{h}C$.*

The proof of Theorem 2 is given in the extended version [15, App. B]. This result justifies our use of joker recipes to represent universal quantification over predicate variables. Indeed, for any $\Phi : \square$ and $\Psi : \varrho$, and a type-compatible Σ , the cell $\langle \perp, \Sigma, \forall g : \varrho. \Phi \rangle$ entails $\langle \perp, \Sigma, (\lambda g : \varrho. \Phi) \Psi \rangle$, and the same holds for their neutralizations.

In the process of evaluation, we can factorize selected first-order monomorphic cells, that is, those that only have term parameters whose type is not `Type`.

Lemma 6. *Consider a cell $C = \langle b, \Sigma, \lambda \bar{x} : (\overline{x:\tau}). \Phi \rangle$ and an aligned stack D, ℓ , such that none of the types τ_i is `Type`. Let D' be the cell $\langle \perp, \emptyset, \lambda \bar{x} : \overline{\tau}. z_1 = x_1 \wedge \dots \wedge z_n = x_n \rightarrow \mathbf{0} \rangle$ for some fresh variables $\bar{z}$. Then $C \circ D, \ell \Leftrightarrow C \circ \mathfrak{h}D, \ell \wedge \forall \bar{z} : \overline{\tau}. (\mathfrak{h}C \circ D', \mathfrak{h}\ell) \vee (D \circ \bar{z})$.*

The proof of Lemma 6 is given in the extended version [15, App. C]. This lemma provides us with an alternative evaluation rule for cell arguments which are eligible and useful to factorize. The latter is a matter of heuristic choice: in our current implementation, we select non-neutral cells that are used multiple times in the final VC and are derived from executable code instead of just a sequence of assertions.

The new rule splits the formula $C \circ D, \ell$ into two parts. The first part, $C \circ \mathfrak{h}D, \ell$, erases all subgoals stemming from D . In the second part, the formula $\mathfrak{h}C \circ D', \mathfrak{h}\ell$ erases all subgoals that are *not* stemming from D , and replaces every occurrence of D with a “unification subgoal” D' , which captures a term substitution in the answer variables $\bar{z}$. These substitutions are transferred to the single instance of D in the formula $D \circ \bar{z}$.

By rewriting the second part as an implication $\forall \bar{z} : \overline{\tau}. \neg(\mathfrak{h}C \circ D', \mathfrak{h}\ell) \rightarrow (D \circ \bar{z})$, we can see the antecedent as the cumulated logical premise (or the strongest postcondition) of the context $C \circ [\] , \ell$ for the continuation in the hole. In the next section, we show how this rule allows us to produce more compact verification conditions.

4 Verification Condition Generation

Verification conditions for COMA expressions are computed by the operator $\mathbb{C}_{\mathfrak{p}}^{\mathfrak{p}}$, where Boolean flags $\mathfrak{p}$ and $\mathfrak{d}$ establish the mode:

- $\mathbb{C}_{\perp}^{\top}$: *caller verification condition*, to verify individual calls of a defined handler.
- $\mathbb{C}_{\top}^{\perp}$: *callee verification condition*, to prove the correctness of a handler definition.
- $\mathbb{C}_{\top}^{\top}$: *full verification condition*, which merges the proof goals of the first two modes.
- $\mathbb{C}_{\perp}^{\perp}$: *null verification condition*, which is always true on fully applied expressions.

The caller mode extracts the specification (or the contract) of a defined handler from its definition. It treats every assertion as a precondition to verify at call sites, and it stops at the black-box barrier which separates the “interface” part of the definition from the hidden “implementation” part. The callee mode, on the contrary, treats every assertion as a precondition to assume, and verifies the correctness of the implementation part, after the black-box barrier, under those assumptions. In the full mode, which is the starting verification mode for COMA expressions, we prove assertions both before and after a barrier. In the null mode, which is equivalent to stopping verification, no proof obligations are generated at all. A COMA program e is said to be *correct*, when its fully evaluated verification condition $\langle \perp, \Sigma_{\text{prim}}, \mathbb{C}_{\top}^{\top}(e) \rangle \circ \varepsilon$ is valid.

$$\begin{aligned}
\mathbb{C}_{\mathfrak{d}}^{\top}(h) &\triangleq h & \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(\pi \rightarrow e) &\triangleq (\lambda \pi. \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(e)) \wedge \mathfrak{h}(\lambda \pi. \mathbb{C}_{\neg \mathfrak{d}}^{\mathfrak{p}}(e)) \\
\mathbb{C}_{\mathfrak{d}}^{\perp}(h) &\triangleq \mathfrak{h}h & \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(k \bar{s} \bar{o}) &\triangleq \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(k) \bar{s} \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(o_1) \dots \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(o_n) \\
\mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(\uparrow e) &\triangleq \mathbb{C}_{\mathfrak{d}}^{\mathfrak{d}}(e) & \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(\{\varphi\} e) &\triangleq (\varphi \rightarrow \mathbb{C}_{\mathfrak{d}}^{\top}(e)) \wedge (\mathfrak{p} \rightarrow \neg \varphi \rightarrow \mathbf{0}) \\
\mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(\downarrow e) &\triangleq \mathbb{C}_{\mathfrak{p}}^{\mathfrak{p}}(e) & \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(e / h \pi = d) &\triangleq \mathbf{let} \ h \ \pi = \mathbb{C}_{\perp}^{\top}(d) \ \mathbf{in} \ \mathbb{C}_{\mathfrak{d}}^{\mathfrak{p}}(e) \wedge \forall \pi. \mathbb{C}_{\mathfrak{p}}^{\perp}(d)
\end{aligned}$$

Fig. 8: Verification condition generation.

Figure 8 shows the rules of VC generation. The notation **let** $h \ \pi = \Psi$ **in** Φ in the rule for handler definitions stands for $(\lambda h : \pi. \Phi) (\lambda \pi. \forall h : \pi. \Psi)$ —notice the universal quantifier that covers the occurrences of h in Ψ and ensures that this symbol is bound in the resulting recipe, just as it is bound in the original COMA expression. In this rule, we assign the handler’s specification $\lambda \pi. \forall h : \pi. \mathbb{C}_{\perp}^{\top}(d)$ to a predicate variable with the same name h . This recipe is verified every time h is called from the underlying expression e or recursively from the definition body d .

Informally, flag $\mathfrak{p}$ determines whether we should generate proof obligations—prove assertions, verify handler definitions, ensure the safety of handler calls—at the current position in the expression. For example, in the rule for $\{\varphi\} e$, we only generate a subgoal for φ (expressed as a double negation $\neg \varphi \rightarrow \mathbf{0}$), when $\mathfrak{p}$ is true. Similarly, in the rule for handler definitions, we verify the correctness of the implementation only when $\mathfrak{p}$ is true; otherwise, the formula $\mathbb{C}_{\perp}^{\perp}(d)$ always reduces to $\top$. Finally, on handler invocation, when $\mathfrak{p}$ is false, the corresponding predicate variable is neutralized, which effectively cancels all proof obligations in the handler’s specification, as $\mathbf{0}$ becomes evaluated as $\top$.

When we pass through a black-box barrier $\uparrow$, the second flag $\mathfrak{d}$ takes the place of $\mathfrak{p}$. Thus, when we compute the specification of a handler by applying $\mathbb{C}_{\perp}^{\top}$ to the handler’s body, we stop at the black-box barrier, where we switch to $\mathbb{C}_{\perp}^{\perp}$, which evaluates to $\top$. On the other hand, when we verify the correctness of a handler definition using $\mathbb{C}_{\perp}^{\perp}$, we do not generate proof obligations for assertions and handler calls until we arrive at the black-box barrier, where we pass into the full mode $\mathbb{C}_{\top}^{\top}$ for the rest of the definition.

The white-box barrier $\downarrow$ replaces the second flag with $\mathfrak{p}$. This preserves the current value of $\mathfrak{p}$ for the rest of the expression, regardless of the subsequent barriers.

The rule for handler calls simply propagates the VC operator down to the individual handlers without changing the mode. Similarly, the rule for anonymous handlers pushes the VC operator in its current mode under the λ -prefix—however, we must, in addition, verify the handler in the complementary mode, with both $\mathfrak{p}$ and $\mathfrak{d}$ negated. This secondary verification condition is only concerned with the continuation parameters of the handler, and not with its proper proof obligations, which is why we neutralize the corresponding recipe. To see why both conditions are necessary, consider the following COMA code:

```
crash / crash = ((f : □) → ↑ f) fail
```

This program reduces to **fail**: we unfold **crash** (E-SYM), purge the now-unreachable definition (E-GC), substitute **fail** into **f** (E-APPH), and drop the barrier (E-BBOX). Thus,

we should not be able to prove it correct. Let us look at the full verification condition:

$$\begin{aligned}
& \mathbb{C}_\top^\top(\text{crash} / \text{crash} = (f \rightarrow \uparrow f) \text{fail}) \\
&= \text{let crash} = \mathbb{C}_\perp^\top((f \rightarrow \uparrow f) \text{fail}) \text{ in } \mathbb{C}_\top^\top(\text{crash}) \wedge \mathbb{C}_\top^\perp((f \rightarrow \uparrow f) \text{fail}) \\
&= \text{let crash} = \mathbb{C}_\perp^\top(f \rightarrow \uparrow f) \mathbb{C}_\perp^\top(\text{fail}) \text{ in } \text{crash} \wedge \mathbb{C}_\top^\perp(f \rightarrow \uparrow f) \mathbb{C}_\top^\perp(\text{fail}) \\
&= \text{let crash} = ((\lambda f. \mathbb{C}_\perp^\top(\uparrow f)) \wedge \mathbb{h}(\lambda f. \mathbb{C}_\top^\perp(\uparrow f))) \text{fail} \text{ in } \text{crash} \wedge \mathbb{C}_\top^\perp(f \rightarrow \uparrow f) (\mathbb{h}\text{fail}) \\
&= \text{let crash} = ((\lambda f. \mathbb{C}_\perp^\top(f)) \wedge \mathbb{h}(\lambda f. \mathbb{C}_\top^\perp(f))) \text{fail} \text{ in } \text{crash} \wedge \mathbb{C}_\top^\perp(f \rightarrow \uparrow f) (\mathbb{h}\text{fail}) \\
&= \text{let crash} = ((\lambda f. \mathbb{h}f) \wedge \mathbb{h}(\lambda f. f)) \text{fail} \text{ in } \text{crash} \wedge \mathbb{C}_\top^\perp(f \rightarrow \uparrow f) (\mathbb{h}\text{fail}) \\
&= \text{let crash} = ((\lambda f. \mathbb{h}f) \wedge \mathbb{h}(\lambda f. f)) \text{fail} \text{ in } \text{crash} \wedge ((\lambda f. f) \wedge \mathbb{h}(\lambda f. \mathbb{h}f)) (\mathbb{h}\text{fail}) \\
&\approx (\lambda f. \mathbb{h}f) \mathbf{0} \wedge (\mathbb{h}\lambda f. f) \mathbf{0} \wedge (\lambda f. f) (\mathbb{h}\mathbf{0}) \wedge (\mathbb{h}\lambda f. \mathbb{h}f) (\mathbb{h}\mathbf{0})
\end{aligned}$$

For the sake of readability, we perform several reductions directly on the recipe in the last step; it is easy to show that the resulting recipe leads to the same final VC formula. Out of the four conjuncts, only the second one evaluates to $\perp$, and the other three to $\top$. If the rule for anonymous handlers did not include the second condition $\mathbb{h}\lambda\pi. \mathbb{C}_{\rightarrow d}^{-p}(d)$, we would end up with only the first and the third conjunct, which both evaluate to $\top$.

Let us further illustrate the rules of VC generation on the example from Sec. 1:

```
triple 14 ((a: int) → { a = 42 } halt) / triple (x: int) (ret r)
= (↑ out (x + x + x)) / out (y: int) = { y = 3 · x } (↑ ret y)
```

The full verification condition of this expression has the form

$$\begin{aligned}
& \text{let triple (x: int) (ret (r: int))} = \mathbb{C}_\perp^\top(e) \text{ in} \\
& \text{triple 14 } ((\lambda a. (a = 42 \rightarrow \text{halt}) \wedge (\top \rightarrow \neg(a = 42) \rightarrow \mathbf{0})) \wedge \\
& \quad \mathbb{h}(\lambda a. (a = 42 \rightarrow \mathbb{h}\text{halt}) \wedge (\perp \rightarrow \neg(a = 42) \rightarrow \mathbf{0}))) \wedge \forall x. \forall \text{ret}. \mathbb{C}_\top^\perp(e)
\end{aligned}$$

where e is the body of `triple`. We can safely drop the secondary verification condition for the anonymous handler in the continuation of `triple`. Indeed, it is produced in the null mode and will evaluate to a tautologically true proof obligation for any argument a .

The caller and callee verification conditions of `triple` are as follows:

$$\begin{aligned}
\mathbb{C}_\perp^\top(e) &= \text{let out (y: int)} = \mathbb{C}_\perp^\top(d) \text{ in } \mathbb{h}\text{out (x + x + x)} \wedge \forall y. \mathbb{C}_\top^\perp(d) \\
\mathbb{C}_\top^\perp(e) &= \text{let out (y: int)} = \mathbb{C}_\perp^\top(d) \text{ in } \text{out (x + x + x)} \wedge \forall y. \mathbb{C}_\perp^\perp(d)
\end{aligned}$$

where d is the body of `out`. And the three VCs of `out` are as follows:

$$\begin{aligned}
\mathbb{C}_\perp^\top(d) &= (y = 3 \cdot x \rightarrow \mathbb{h}\text{ret y}) \wedge (\top \rightarrow \neg(y = 3 \cdot x) \rightarrow \mathbf{0}) && \approx y = 3 \cdot x \\
\mathbb{C}_\top^\perp(d) &= (y = 3 \cdot x \rightarrow \text{ret y}) \wedge (\perp \rightarrow \neg(y = 3 \cdot x) \rightarrow \mathbf{0}) && \approx y = 3 \cdot x \rightarrow \text{ret y} \\
\mathbb{C}_\perp^\perp(d) &= (y = 3 \cdot x \rightarrow \mathbb{h}\text{ret y}) \wedge (\perp \rightarrow \neg(y = 3 \cdot x) \rightarrow \mathbf{0}) && \approx \top
\end{aligned}$$

On the right we show the first-order formulas to which these recipes evaluate. Notice that for any predicate `ret`, the recipe $\mathbb{h}\text{ret y}$ evaluates to a tautology.

Now we can evaluate the verification conditions of `triple`. Since $\mathbb{h}\text{out}$ evaluates to a tautology for any argument, the caller VC, $\mathbb{C}_\perp^\top(e)$, evaluates to $\forall y. y = 3 \cdot x \rightarrow \text{ret y}$. And the callee VC, $\mathbb{C}_\top^\perp(e)$, evaluates to $x + x + x = 3 \cdot x$. After simplification, the full VC of the initial expression is $(\forall y. y = 3 \cdot 14 \rightarrow y = 42) \wedge (\forall x. x + x + x = 3 \cdot x)$.

Verification conditions produced by the $\mathbb{C}_+^\top$ operator are for partial correctness: they do not ensure the termination of COMA programs. Here is one possible way to verify total correctness. Let us say that a handler definition $f \bar{x}:\bar{\tau} \ \bar{g}:\bar{\varrho} = d$ is equipped with a *variant*, if there exists an int-typed term $t[\bar{x}]$ such that every occurrence of f in d is in an expression of the form $\{t[\bar{s}] < t[\bar{x}] \wedge \emptyset \leq t[\bar{x}]\} f \bar{s} \bar{o}$. Here we assume that variables are not bound twice, so that the variables $\bar{x}$ in the assertion refer to the formal parameters of f . The ordering relation in the assertion is well-founded, therefore, an infinite tower of recursive calls of f is impossible. A practical implementation would, of course, accept other well-founded relations, such as structural decrease on binary trees, lexicographic orderings on tuples, etc. While the definition above does not allow us to use f as a handler argument inside d , this is not a limitation, as we can always move such occurrences into a local wrapper handler definition.

Below we list the main results about our VC generation procedure. The proofs are given in the extended version [15, Appendix D].

Lemma 7. *For any COMA expression e , any cell of the form $\langle b, \Sigma, \mathbb{C}_\perp^\perp(e) \rangle$ is neutral.*

Consequently, any VC of the form $\mathbb{C}_\perp^\perp(e)$, where e is a fully applied expression, can be safely replaced with a tautological recipe such as $\mathfrak{h}\mathbf{0}$.

Lemma 8. *For any COMA expression e , any cell of the form $\langle b, \Sigma, \mathbb{C}_+^\top(e) \rangle$ is a meet of $\langle b, \Sigma, \mathbb{C}_\mathfrak{p}^\mathfrak{p}(e) \rangle$ and $\langle b, \Sigma, \mathbb{C}_\mathfrak{d}^\mathfrak{d}(e) \rangle$ for all $\mathfrak{p}$ and $\mathfrak{d}$.*

The fact that $\mathbb{C}_+^\top(e)$ can be split into $\mathbb{C}_\perp^\top(e)$ and $\mathbb{C}_+^\perp(e)$ is the basis of the correctness preservation theorem:

Theorem 3 (Preservation of Correctness). *For any COMA programs e and e' , if e is correct and $e \longrightarrow e'$, then e' is correct.*

Theorem 4 (Progress). *For any correct COMA program e , either e is halt , or $e \longrightarrow e'$ for some program e' .*

In conclusion, let us show some examples of verification conditions. For clarity, we omit type annotations, inline the specifications of primitive handlers, treat $\mathbf{0}$ as $\perp$ in the callee mode, and remove trivial subgoals coming from $\mathbb{C}_\perp^\perp(\cdot)$ or $\perp \rightarrow \Phi$. The caller VC of `removeRoot` on page 183, for a given tree t and continuation `return`, is the recipe

$$\begin{aligned} (\forall lvr. t = \text{Node } l \vee r \rightarrow \forall s. (\forall e. e \in s \leftrightarrow e \in l \vee e \in r) \rightarrow \text{return } s) \wedge \\ (t = \text{Empty} \rightarrow \mathbf{0}) \end{aligned}$$

This recipe is the specification of `removeRoot`, instantiated and proved at each call site. The subrecipe $(\forall e. e \in s \leftrightarrow e \in l \vee e \in r) \rightarrow \text{return } s$ is the callee VC for the out handler. Notice that the assertion $\{\forall e:\text{int}. e \in s \leftrightarrow e \in l \vee e \in r\}$ does not generate a subgoal here: as it occurs before the black-box barrier, it is treated as an assumption in the callee mode.

Here is the callee VC for `removeRoot`, to be proved for all values of t :

$$\forall lvr. t = \text{Node } l \vee r \rightarrow \text{mergeTree } elt \ l \ r \ (\lambda s. \forall e. e \in s \leftrightarrow e \in l \vee e \in r)$$

There is no subgoal generated for the second outcome of `unTree`, as it does not contain an abstraction barrier. The predicate argument of `mergeTree` is the caller VC of `out`; this time the assertion does generate a subgoal.

Here is the specification of the product handler in Fig. 5, for given integers a , b and a continuation `return`:

$$(b \neq 0 \rightarrow 0) \wedge (\forall c. c = a \cdot b \rightarrow \text{return } c)$$

The callee VC for `product`, to be proved for all values of a and b , is as follows:

$$\begin{aligned} & b \geq 0 \rightarrow \\ & a \cdot b + 0 = a \cdot b \wedge b \geq 0 \wedge \\ & \forall pqr. p \cdot q + r = a \cdot b \wedge q \geq 0 \rightarrow \\ & (q > 0 \rightarrow \\ & (q \bmod 2 = 1 \rightarrow (p + p) \cdot (q \text{ div } 2) + r + p = a \cdot b \wedge q \text{ div } 2 \geq 0) \wedge \\ & (q \bmod 2 \neq 1 \rightarrow (p + p) \cdot (q \text{ div } 2) + r = a \cdot b \wedge q \text{ div } 2 \geq 0)) \wedge \\ & (q \neq 0 \rightarrow r = a \cdot b) \end{aligned}$$

Notice how this formula coincides with the verification condition for the definition of `product` obtained by the traditional weakest-precondition calculus. Consider now the same VC, when we select the next handler for factorization, as described in Section 3:

$$\begin{aligned} & b \geq 0 \rightarrow \\ & a \cdot b + 0 = a \cdot b \wedge b \geq 0 \wedge \\ & \forall pqr. p \cdot q + r = a \cdot b \wedge q \geq 0 \rightarrow \\ & (q > 0 \rightarrow \\ & \quad \forall s. ((q \bmod 2 = 1 \wedge s = r + p) \vee (q \bmod 2 \neq 1 \wedge s = r)) \rightarrow \\ & \quad (p + p) \cdot (q \text{ div } 2) + s = a \cdot b \wedge q \text{ div } 2 \geq 0) \wedge \\ & (q \neq 0 \rightarrow r = a \cdot b) \end{aligned}$$

The formula $\lambda s. (q \bmod 2 = 1 \wedge s = r + p) \vee (q \bmod 2 \neq 1 \wedge s = r)$ is the strongest postcondition of the expression `if (q mod 2 = 1) (→ next (r + p)) (→ next r)` with respect to the continuation `next`. The method of compact verification conditions proposed by Flanagan and Saxe [7] aggregates in a similar way the strongest postconditions across alternative execution paths. The connection between the compact verification conditions and the classical weakest-precondition calculus was studied by Leino [12]. Our approach makes this connection even more prominent, as it allows us to derive both forms from the common precursor verification condition.

5 Implementation

We have implemented the `COMA` language and its VC generator on top of the `WHY3` platform [6]. The terms and formulas of `COMA` are written in the logical language of `WHY3`. This way, we can make use of logical theories from the `WHY3` standard library, and we readily benefit from `WHY3`'s interface with many automated theorem provers. In addition to what is presented in the previous sections, our implementation offers a few extensions, described below.

```

product (a b: int) {  $b \geq 0$  } (return (c: int) {  $c = a \cdot b$  })
= loop a b 0
  / loop (p q r: int) {  $p \cdot q + r = a \cdot b \wedge q \geq 0$  }
    = if (q > 0) (→ if (q mod 2 = 1) (→ next (r + p)) (→ next r)
      / next (s: int) = loop (p + p) (q div 2) s)
      (→ return r)

```

Fig. 9: Extended handler prototypes.

Extended handler prototypes. To facilitate writing and understanding of COMA programs, we provide a suitable syntax for writing pre- and postconditions directly in the handler prototype, as shown in Fig. 9. This notation is desugared into assertions, barriers, and wrapper handlers of the core COMA language; in particular, the code in Fig. 9 is translated into what is shown in Fig. 5. The precondition $\{b \geq 0\}$ in the prototype of `product` becomes an assertion put on top of the definition body, now hidden under a black-box barrier. The same transformation is applied to the precondition of the inner loop handler. The postcondition $\{c = a \cdot b\}$, attached to `return`, forces creation of a wrapper handler above the main black-box barrier and becomes the precondition in the body of this wrapper handler.

Let-binding for variables. We added a proper syntax for binding a variable to a term, to avoid writing anonymous handler applications $((x:\tau) \rightarrow e) s$. The new construction is written $e / x:\tau = s$. We show its use in the example in Fig. 12 in the next section.

Mutable state. Our implementation supports mutable variables (*references*) that can be allocated, modified, and passed as arguments to handlers. References are alias-free, which means that is forbidden to pass a statically accessible reference as an argument or to pass the same reference argument twice. Each handler is annotated with a *pre-write* clause, which lists the references in its lexical scope that might be modified before the handler is executed. For example, here is the prototype of a handler that increments an integer reference received as argument and returns its previous value:

```
incr (&r: int) (return [r] (p: int))
```

The pre-write annotation `[r]` for the `return` outcome signifies that the code that calls `return`—namely, the `incr` handler—may change the value of `r` before the call. Pre-write annotations are automatically inferred for defined handlers and their continuation parameters. However, we do not infer them for the higher-order outcomes (i.e., continuation parameters of a continuation parameter).

The code with references is translated into pure COMA via a fine-grained monadic transformation, during which the references in the pre-write annotations become additional term parameters. In the example above, after translation, `incr` would return to the caller the updated value of `r` along with its previous value in `p`.

To capture the pre-state of references, we admit let-bindings in handler prototypes. The full prototype of `incr`, together with its specification, is as follows:

```
incr (&r: int) [o: int = r] (return [r] (p: int) {  $r = o + 1 \wedge p = o$  })
```

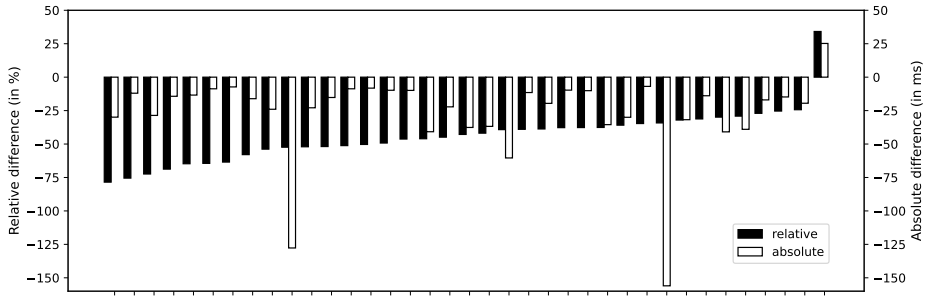


Fig. 10: VC generation time: COMA vs WHY3 (lower values favor COMA).

Specification extraction. Given a first-order handler, COMA can produce, on request, the logical predicates that represent its pre- and postconditions. Consider the following handler which returns the value stored in the root node of a non-empty tree:

```
getRoot (t: tree elt) (return (v: elt)) =
  unTree elt t ((_: tree elt) (u: elt) ( _: tree elt) → return u) fail
```

The caller VC of `getRoot`, for a given tree t and continuation `return`, is the recipe

$$(\forall lvr. t = \text{Node } l \vee r \rightarrow \text{return } v) \wedge (t = \text{Empty} \rightarrow \mathbf{0}) \quad (\star)$$

To compute the precondition of `getRoot`, we merely need to instantiate `return` with a neutral recipe $\lambda x. \mathbf{h0}$. After evaluation and simplification, we obtain $t \neq \text{Empty}$.

To compute the postcondition, we neutralize the recipe $(\star)$, eliminating the subgoals that belong to the precondition, and instantiate `return` with the recipe $\lambda x. z = x \rightarrow \mathbf{0}$, where z is fresh. After evaluation, we obtain $\forall lvr. t = \text{Node } l \vee r \rightarrow z = v \rightarrow \perp$, which is equivalent to $\neg(\exists l r. t = \text{Node } l \vee r)$. This last formula is exactly the negated postcondition of `getRoot`, where z denotes the returned value.

This procedure works in a similar way to subgoal factorization discussed in Sec. 3. It is easily extended to handlers with multiple outcomes, producing a separate postcondition for each of them.

Benchmarks. We evaluated the performance of our VC generator using the CREUSOT test suite². CREUSOT switched its VC generation back-end from WHY3 to COMA at release 0.2. We used the Rust files from the test suite that did not change during the switch (235 files). For each of those files, we measured the VC generation time with both back-ends and excluded those where the absolute difference was smaller than the sum of two standard deviations. For the 33 remaining files, Fig. 10 shows the relative and the absolute time difference (black and white bars, respectively). The negative values in the chart are where the VCgen of COMA is faster than that of WHY3, which is the case for all but one test file. For a third of the tests, COMA was more than 50% faster.

² This evaluation can be reproduced using the instructions in the Zenodo archive available online at <https://doi.org/10.5281/zenodo.14766822>.

```

type regexp = Empty      | Char of char      | Alt    of regexp * regexp
              | Epsilon   | Star of regexp   | Concat of regexp * regexp

let accept (r: regexp) (s: string): bool =
  let n = String.length s in
  let rec a (r: regexp) (i: int) (k: int → unit): unit = match r with
    | Empty      → ()
    | Epsilon    → k i
    | Char c     → if i < n && s.[i] = c then k (i + 1)
    | Alt (r1, r2) → a r1 i k; a r2 i k
    | Concat (r1, r2) → a r1 i (fun j → a r2 j k)
    | Star r1     → k i; a r1 i (fun j → if i < j then a r j k) in
  try a r 0 (fun j → if j = n then raise Exit); false with Exit → true

```

Fig. 11: Regular expression engine in OCaml.

```

accept (r: regexp) (s: string) (return (b: bool))
= a r 0 (j h → if (j = n) (→ return true) h) (→ return false)
  / a (r: regexp) (i: int) (k (j: int) (h)) (o)
  = unRe r empty eps char alt cat star
    / empty      = o
    / eps        = k i o
    / char c     = if (i < n && s[i] = c) (k (i + 1) o) o
    / alt r1 r2  = a r1 i k (→ a r2 i k o)
    / cat r1 r2  = a r1 i (j h → a r2 j k h) o
    / star r1    = k i (→ a r1 i (j h → if (i < j) (→ a r j k h) h) o)
  / n: int = length s

```

Fig. 12: Regular expression engine in COMA.

6 Case Study: Regular Expression Processing

In this section, we demonstrate the use of COMA by verifying a small, yet non-trivial OCaml program, that uses higher-order functions, exceptions, and requires giving specification to closures in order to be verified. Figure 11 shows a function `accept` that checks if a string `s` is recognized by a regular expression `r`. The code traverses the string with a recursive function `a`, which takes three parameters: a current regexp `r`, an integer index `i`, and a continuation `k`. This function tries to match a substring `s[i..j]` with `r`, and then applies the continuation `k` to index `j` to proceed with the matching of `s[j..]`. If no such `j` exists, function `a` returns the unit value `()`. The initial continuation passed to function `a` signals a success by raising the predefined exception `Exit`.

Figure 12 contains a COMA version of the `accept` function, which can be obtained by a mechanical CPS-translation of the OCaml code. The `accept` handler has a return outcome that receives the Boolean computation result. The local handler `a` has a continuation `o`, that corresponds to the normal outcome of the original OCaml function.

```

predicate cons (s: string) (r: regexp) (ck: int → bool) (i: int) =
  exists j. i ≤ j ≤ length s ∧ mem s[i..j] r ∧ ck j

accept (r: regexp) (s: string) (return (b: bool) { b ↔ mem s r })
= a r 0 (j ↦ j = n) (j h → if (j = n) (→ return true) h) (→ return false)
/ a (r: regexp) (i: int) (ck: int → bool) { 0 ≤ i ≤ n }
  (k (j: int) { mem s[i..j] r ∧ i ≤ j ≤ n } (h { not (ck j) }))
  (o { not (cons s r ck i) })
= unRe r empty eps char alt cat star
  / empty      = o
  / eps        = k i o
  / char c     = if (i < n && s[i] = c) (k (i + 1) o) o
  / alt r1 r2  = a r1 i ck k (→ a r2 i ck k o)
  / cat r1 r2  = a r1 i (j ↦ cons s r2 ck j) (j h → a r2 j ck k h) o
  / star r1    = k i (→ a r1 i (j ↦ i < j ∧ cons s r ck j)
                    (j h → if (i < j) (→ a r j ck k h) h) o)

/ n: int = length s

```

Fig. 13: Regular expression engine in CoMA, with specification.

Finally, the continuation k is transformed itself into CPS-style, and thus has its own outcome, named h . Another way to look at this code is to interpret k and o as success and error/backtrack continuations, respectively, as in a double-barreled CPS [18]. The pattern-matching on the regular expression r is performed using a library handler `unRe` similar to the `unTree` handler.

To verify this program, we need to add specifications. Figure 13 contains a version of `accept` with added preconditions (in cyan) and ghost parameters (in gray). As explained in Section 5, specification annotations in handler prototypes are automatically desugared into assertions, black-box barriers, and wrapper handlers. The postcondition of `accept` (i.e., the precondition of `return`) uses a built-in logical predicate `mem`, where `mem s r` holds if and only if the string s belongs to the language of r .

We add a ghost parameter `ck` to handler `a` for the purpose of its specification. Note that the current implementation of CoMA does not provide any special treatment for ghost code and data. In future, we plan to introduce the necessary checks that ensure that ghost computations do not interfere with the observable part of the program. As in the OCaml code, handler `a` tries to match a substring $s[i..j]$ with r , and then applies the continuation k to index j to proceed with the matching of $s[j..]$. The `ck` parameter is a first-class predicate which characterizes the index j passed to k . The `cons` predicate, declared on top of the figure, is a shortcut to simplify annotations.

In addition to the annotations given in Fig. 13, we have also instrumented the CoMA code to verify the termination of handler `a`, as described in Section 4. In this case, the variant is a pair $(|s| - i, r)$, ordered lexicographically: namely, we either progress in string s or we move to a smaller regular expression. When the VC for handler `accept` is sent to WHY3, it is split into 44 individual proof tasks which are easily discharged by the SMT solvers Z3 [4] and Alt-Ergo [3].


```

# sort the bits of %rdi using ``I can't believe it can sort''
sortbits:
    mov    %rdi, %rax
    mov    $0x8000000000000000, %rdi
loop1:   #@ invariant pop(rdi) = 1  $\wedge$  pop(rax) = pop(rdi@sortbits)
    mov    $0x8000000000000000, %rsi
loop2:   #@ invariant pop(rsi) = 1  $\wedge$  pop(rax) = pop(rdi@sortbits)
    mov    %rax, %rcx                # if !(rax & rdi)
    and    %rdi, %rcx
    jnz    cont2
test2:   mov    %rax, %rcx            # and (rax & rsi)
    and    %rsi, %rcx
    jz     cont2
swap:    or     %rdi, %rax            # then swap bits
    andn   %rax, %rsi, %rax
cont2:   shr    $1, %rsi
    jnz    loop2
cont1:   shr    $1, %rdi
    jnz    loop1
    #@ assert pop(rax) = pop(rdi@sortbits)
    ret

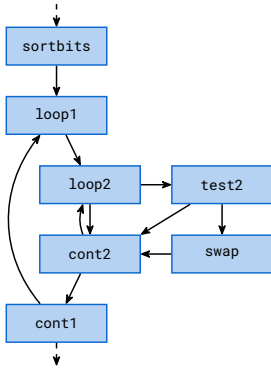
```

Fig. 14: Example of verified x86-64 code.

7 Case Study: Verified Assembly Code

We believe that COMA is a suitable intermediate language for the verification of unstructured programs. As a proof of concept, we have built a prototype tool for the deductive verification of x86-64 assembly programs. The input of the tool is assembly code annotated with assumptions, assertions, and loop invariants. Figure 14 shows the x86-64 assembly code for a function `sortbits` that sorts the bits of a 64-bit integer using the “I can’t believe it can sort” algorithm by Fung [8]. We use the AT&T syntax, with the destination operand on the right. For instance, `mov A, B` copies the register `A` into `B`, and `andn A, B, C` computes $A \wedge \neg B$ and stores it in `C`. Integer literals start with a `$` sign. The code contains unnecessary labels (e.g., `test2`), which are only introduced to simplify the forthcoming explanations.

Function `sortbits` receives an integer in the `rdi` register and returns an integer in the `rax` register, with the same number of 1 bits, which are moved to the least significant positions. The code iterates over all pairs of bits $0 \leq i, j < 64$, using registers `rdi` and `rsi`, with two nested loops. Whenever the bit i is clear and the bit j is set, the two bits are swapped. (It is not obvious why this sorting procedure is correct; see Fung’s paper for an explanation.) The code contains logical annotations as special comments: namely, two loop invariants and one assertion before the function end. Here, we only show that the population count remains constant (using a logical function `pop`), but we do not show that bits are indeed sorted. We use the notation `rdi@sortbits` to refer to the value of the `rdi` register at function entry.



```

sortbits &rax &rcx &rsi &rdi &flags [rdi0=rdi]
= [...]
loop1
/ loop1 =
{ pop(rdi) = 1 ∧ pop(rax) = pop(rdi0) }
(↑ [...] loop2
/ loop2 =
{ pop(rsi) = 1 ∧ pop(rax) = pop(rdi0) }
(↑ [...] if  $b_0$  cont2 test2
/ test2 =
([...] if  $b_1$  cont2 swap
/ swap = [...] cont2)
/ cont2 =
([...] if  $b_2$  loop2 cont1
/ cont1 =
([...] if  $b_3$  loop1 exit
/ exit = { pop(rax) = pop(rdi0) }
halt))))

```

(a) Control flow of sortbits.

(b) Structure of the COMA code for sortbits.

Fig. 15: Compilation passes for sortbits.

Our tool parses the code and its annotations, and starts with building its control-flow graph (depicted in Fig. 15a). Then, it computes the dominator tree, the entry point being the function entry. A basic block A dominates a block B whenever any path from the entry to B traverses A . For instance, block loop2 dominates block test2 , which itself dominates swap2 . Finally, our tool builds a COMA code that follows the structure of the dominator tree. In this way, we do not need to repeat the outer invariant in the inner loop for variables that are not modified. For instance, the handler swap2 is a local definition in handler test2 , which is itself a local definition in handler loop2 . Each invariant is translated into an assertion followed by a barrier. Figure 15b shows the structure of the COMA code for `sortbits`. For an easier reading, we omit type annotations and we have left only what relates to control-flow and specification. Conditional jumps are translated using the primitive `if` (and suitable Boolean conditions b_i), and unconditional jumps are handler calls. Parts where references are modified are written “[...]” for clarity. The translated code relies on our WHY3 model of a fragment of the x86-64 instruction set. For instance, the instruction `andn rax, rsi` is translated into a COMA reference assignment $\&\text{rax} \leftarrow \text{andn } \text{rax } \text{rsi}$, where the logical function `andn` is defined in the accompanying WHY3 library.

The COMA back-end computes the VC for the code in Fig. 15b, and sends it to WHY3. There it is split into 5 proof tasks—two instances of invariant initialization, two instances of invariant preservation, and the final postcondition—which are automatically proved by Z3 and Alt-Ergo.

8 Related work

To our knowledge, no deductive verification system features explicit abstraction barriers in the style of COMA. Compared to the widely used intermediate verification languages like BOOGIE [13], VIPER [14] or WHYML [6], COMA is a smaller language, with fewer constructs and a simpler VC generator. Still, we are not aware of an autoactive program verifier that would admit the higher-order regex example from Section 6. To handle stateful computations, COMA implements first-class alias-free mutable variables, which is close to what is provided by BOOGIE and WHYML. VIPER, in comparison, offers a significantly more powerful ownership-based reasoning framework.

There is a natural connection between the weakest-precondition calculus and the CPS transformation, the former being a predicate transformer and the latter a structurally similar code transformer. This connection was first studied on a minimal imperative language by Jensen [9]. This work was later extended with exception handling and goto statement by Audebaud and Zucca [1], and furthermore, with recursion, higher-order functions, and side effects by Kura [11]. A predicate transformer called the Dijkstra monad, introduced by Swamy et al. [17] and used to verify higher-order and effectful F^* programs, also highlights this connection. COMA exploits in a similar manner the relation between the continuation-based style and the WP computation. Explicit abstraction barriers allow us to verify recursive code without computing a fixed point of its verification condition.

The CFML tool developed by Charguéraud [2] serves for interactive verification of higher-order stateful programs, written in a subset of OCaml. Programs are translated into so-called characteristic formulas, which essentially capture the weakest precondition of the programs, with respect to a shallow embedding of separation logic in Coq. Specifications are proved in the form of lemmas derived from characteristic formulae. Unlike COMA, or other VC-based program verifiers, the program logic rules of CFML have to be applied manually in the course of an interactive Coq proof. Assertions and invariants are provided as the proof progresses, which limits the possibilities for automation. On the other hand, CFML offers a greater flexibility in stating properties of program code, such as verifying a given function against two different contracts.

9 Conclusion

We presented COMA, a higher-order IVL with explicit abstraction barriers that can be placed inside function definitions to make the exposed part of the computation appear in the specification. This allows us to write specifications in a more concise and flexible way, without having to manually translate executable code into logical specification.

In the future, we consider adding the mechanisms of ownership and borrowing to handle the mutable state, and using prophecy variables in verification conditions. We are also interested in supporting more advanced control structures, such as iterators, coroutines, and algebraic effects. We consider an alternative VC rule, where an above-barrier subhandler definition is verified at the callee site rather than the caller site. In certain programs with nested loops, like the `sortbits` example from Sec. 7, this allows us to move all invariants into inner loops, avoiding repetition and reducing specification burden. Finally, we continue to improve the efficiency of our implementation, in particular the recipe evaluation engine and the subgoal factorization heuristics.

References

1. Audebaud, P., Zucca, E.: Deriving proof rules from continuation semantics. *Formal aspects of computing* **11**, 426–447 (1999)
2. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. vol. 46(9), pp. 418–430. ACM, Tokyo, Japan (September 2011)
3. Conchon, S., Coquereau, A., Iguernelala, M., Mebsout, A.: Alt-Ergo 2.2. In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories* (2018)
4. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. vol. 4963, pp. 337–340. Springer (2008)
5. Denis, X., Jourdan, J.-H., Marché, C.: Creusot: a foundry for the deductive verification of Rust programs. In: *International Conference on Formal Engineering Methods*. vol. 13478, pp. 90–105. Springer (2022)
6. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) *Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science*, vol. 7792, pp. 125–128. Springer (Mar 2013)
7. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: *Principles Of Programming Languages*. vol. 36(3), pp. 193–205. ACM (2001)
8. Fung, S.P.Y.: Is this the simplest (and most surprising) sorting algorithm ever? *arXiv preprint arXiv:2110.01111* (2021)
9. Jensen, K.: Connection between Dijkstra’s predicate-transformers and denotational continuation-semantics. *DAIMI Report Series* **86** (1978)
10. Krivine, J.-L.: *Un interpréteur du λ -calcul*. Unpublished report (1985)
11. Kura, S.: Higher-Order Weakest Precondition Transformers via a CPS transformation. *arXiv preprint arXiv:2301.09997* (2023)
12. Leino, K.R.M.: Efficient weakest preconditions. *Information Processing Letters* **93**(6), 281–288 (2005)
13. Leino, K.R.M.: This is Boogie 2. *KRML Manuscript* **178** (2008)
14. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: *Verification, Model Checking, and Abstract Interpretation: 17th International Conference*. vol. 9583, pp. 41–62. Springer (2016)
15. Paskevich, A., Patault, P., Filliâtre, J.-C.: Coma, an Intermediate Verification Language with Explicit Abstraction Barriers (extended version). *Tech. rep.* (2025), <https://hal.science/hal-04839768>
16. Statman, R.: Logical relations and the typed λ -calculus. *Information and control* **65**(2-3), 85–97 (1985)
17. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. *ACM SIGPLAN Notices* **48**(6), 387–398 (2013)
18. Thielecke, H.: Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation* **15**, 141–160 (2002)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





A Complete Axiomatisation of Equivalence for Discrete Probabilistic Programming

Robin Piedeleu¹, Mateo Torres-Ruiz^{1(✉)}, Alexandra Silva², and Fabio Zanasi¹

¹ University College London, London, UK
r.piedeleu@ucl.ac.uk, m.torresruiz@cs.ucl.ac.uk

² Cornell University, Ithaca, USA

Abstract. We introduce a sound and complete equational theory capturing equivalence of discrete probabilistic programs, that is, programs extended with primitives for Bernoulli distributions and conditioning, to model distributions over finite sets of events. To do so, we translate these programs into a graphical syntax of probabilistic circuits, formalised as string diagrams, the two-dimensional syntax of symmetric monoidal categories. We then prove a first completeness result for the equational theory of the conditioning-free fragment of our syntax. Finally, we extend this result to a complete equational theory for the entire language. Our first result gives a presentation of the category of Markov kernels, restricted to objects that are powers of the two-elements set.

1 Introduction

Probabilistic programming languages (PPLs) extend standard languages with two added capabilities: drawing random values from a given distribution, and conditioning on particular variable values. These constructs allow the programmer to define complex statistical models and obtain the probability of an event according to the distribution specified by the program, a task known as *inference*. This process can be understood in Bayesian terms: first-class distributions define a prior over the program’s declared variables; inference then involves computing the posterior distribution, conditional on some chosen values of those variables.

In this paper, we focus on *discrete* probabilistic programs, that is, programs whose associated probability distributions range over a finite set of outcomes. While many existing PPLs manipulate continuous random variables, this generally forces them to approximate the posterior distribution specified by a given program, typically by sampling from it [39,28,9]. Discrete probabilistic programs on the other hand lend themselves to *exact* inference, where instead of approximating or sampling from the posterior distribution, it is possible to obtain an exact representation of it, which is interesting for verification purposes.

Several existing languages support this style of inference for discrete probabilistic programs [22,7,18]. In this work, we start from a prototypical discrete

R. Piedeleu and M. Torres-Ruiz—Both authors contributed equally to this research.

PPL that incorporates a primitive for Bernoulli distributions (`flip`) and conditioning ($x := y$). Beyond its probabilistic features, its syntax is that of a simple first-order, non-recursive functional programming language with `let`-expressions for variable declaration and branching with an `if-then-else` constructor. In this respect, it is similar to *Dice* [22] or the *CD-calculus* [36]. While it may appear limited, it is in fact sufficiently expressive to specify arbitrary distributions over tuples of Boolean variables and thus, to encode arbitrary discrete distributions.

The example on the right expresses a simple model, the *von Neumann trick*, as a probabilistic program. Imagine that we are betting on the outcome of a coin flip and cannot trust that the coin is fair. We can use the following protocol to simulate a fair coin: toss the coin twice (`flip p`); if the outcomes are the same (`((first := (not second))`), discard them and start over; otherwise, keep the first outcome and discard the second. This is exactly the program above, where p is an arbitrary parameter strictly between 0 and 1, the unknown probability that the coin lands on heads.

As this example illustrates, probabilistic programs are a compact way to write and communicate probabilistic models: they are modular, compositional, and understood by a large community of practitioners, who can then leverage functional abstraction to build and share increasingly complicated models.

A probabilistic program can often be replaced by an *equivalent* program whose corresponding inference task is simpler to solve. We say that two probabilistic programs are equivalent when they define the same distribution. Von Neumann’s trick provides a first example of a non-trivial program equivalence: the program above is interchangeable with the much simpler program `flip 0.5`.

This notion of equivalence also applies to *partial* programs containing free variables, like the one on the left which is equivalent to the identity function on a Boolean variable.

Program equivalence is essential to ensure that the behaviour of programs remain consistent across different implementations and refinements [24,21]. In probabilistic programming, equivalence checking is crucial for building reliable, efficient, and accurate models, as well as inference algorithms. However, probabilistic features greatly complicate the task of verifying program equivalence.

In this work, we tackle the equivalence problem by presenting a complete equational theory for discrete probabilistic programs. To do so, we translate the simple PPL syntax above into an equally expressive diagrammatic calculus of *probabilistic circuits*. We start from plain Boolean circuits with standard logical gates, and extend them with two additional gates that incorporate probabilistic features: one representing a Bernoulli distribution, another conditioning.

Note that the behaviour of the `if-then-else` construct used in this paper differs from the standard interpretation found in most programming languages. Although our interpretation is natural from the perspective of (probabilistic) circuits, it allows failure in non-executed branches to propagate, breaking equivalences expected to hold in branching constructs (i.e., `(if true then A else B) = A` and `(if false then A else B) = B` will not hold for all A s and B s).

While this behaviour aligns with logical circuits when no failures occur, defining its operational semantics explicitly would require an unusual approach. As we will see, though atypical, this behaviour is a necessary consequence of our choice of semantics, which equates programs who encode the same posterior.

While circuits are often used as informal visual aids, the circuits in this paper are a formal syntax of *string diagrams*, the graphical language of (symmetric) monoidal categories [34]. In recent years, string diagrams have been applied in different fields, including quantum computing [8], electrical engineering [4] and probability theory [13]. They offer several advantages: like programs, we can reason about them algebraically, as syntactic objects with sequential and parallel compositions, and endow them with a compositional semantics; like circuits, the visual representation absorbs structural equivalences of programs and highlights resource-exchange between the different parts of the systems they represent.

The main technical result of our paper is proving that the provided equational theory fully characterises semantic equality of circuits: we show that whenever two circuits denote the same distribution, we can derive this fact using purely equational reasoning on the circuits themselves. To prove this result, we proceed in two steps: we first give a complete equational theory for the conditioning-free fragment of our syntax, which we later extend to the whole language.

Note that these results can be seen as part of a broader program aiming to formulate probability theory in *Markov categories* [13]. In particular, our first main result provides a presentation in terms of generators and equations of the full monoidal subcategory of the category of Markov kernels/stochastic maps [19] on objects that are powers of the two-element set ($\mathbb{B}^n$ for some natural number n). Our contribution is thus one further step on the way to developing an axiomatic account of probability theory in Markov categories.

Outline and main contributions. In Section 2, we give the formal syntax of probabilistic circuits, both as a standard term syntax and as a two-dimensional syntax of string diagrams; we explain its relation with a conventional language for discrete probabilistic programming such as the one used in the previous examples; and we equip it with a compositional semantics which captures the distribution the circuits are intended to represent. In Section 3 we introduce and prove the completeness of an equational theory for the conditioning-free fragment of our circuits. In Section 4 we extend this theory to the full syntax, and prove it complete, thus obtaining a full axiomatisation of discrete probabilistic program equivalence. Section 5 discusses related work and directions for future work.

2 Syntax and Semantics

In this section, we introduce the diagrammatic notation used as syntax for probabilistic programs and define semantics in terms of (sub)distributions.

2.1 Syntax

We define our syntax in two steps: we start with a simple grammar whose constants are circuit primitives with rules for their sequential and parallel composi-

$$\begin{array}{c}
\bullet : 1 \rightarrow 0 \quad \bullet \text{---} \text{---} : 1 \rightarrow 2 \quad \text{---} \text{---} : 2 \rightarrow 1 \quad \text{---} \text{---} : 1 \rightarrow 1 \\
\text{---} \text{---} : 0 \rightarrow 1 \quad \text{---} \text{---} : 2 \rightarrow 1 \quad \text{---} : 0 \rightarrow 0 \quad \text{---} : 1 \rightarrow 1 \quad \text{---} : 2 \rightarrow 2 \\
\frac{c : \ell \rightarrow m \quad d : m \rightarrow n}{c; d : \ell \rightarrow n} \quad \frac{c : m_1 \rightarrow n_1 \quad d : m_2 \rightarrow n_2}{c \otimes d : m_1 + m_2 \rightarrow n_1 + n_2}
\end{array}$$

Fig. 1: Typing rules for ProbCirc terms. A type is a pair $(m, n) \in \mathbb{N} \times \mathbb{N}$ which we write as $m \rightarrow n$; we write the judgment that c has type $m \rightarrow n$ as $c : m \rightarrow n$.

tion, and then move to a two-dimensional representation of the same circuits as string diagrams. We compare our graphical language with a more conventional programming language syntax, by providing a syntactic translation.

Term language. We consider terms given by the following simple grammar:

$$\begin{array}{l}
\text{ProbCirc} \ni c ::= \bullet \mid \bullet \text{---} \text{---} \mid \text{---} \text{---} \mid \text{---} \text{---} \mid \text{---} \mid \text{---} \mid \text{---} \mid \text{---} \mid c; c \mid c \otimes c \quad (1) \\
\mid \text{---} \text{---} \quad \forall p \in [0, 1] \quad (2) \\
\mid \text{---} \text{---} \quad (3)
\end{array}$$

Though the constants of our language are depicted as diagrams, we treat them as symbols for the moment. Soon, we will consider all ProbCirc-terms as *string diagrams* and view the two binary operations of *sequential* ($c; d$) and *parallel* ($c \otimes d$) composition as those of a monoidal category [34].

We will refer to the fragment of the syntax which contains the rules and constants of line (1) as BoolCirc, and the syntax of lines (1) and (2) as CausCirc.

Our language does not have variables, making it unnecessary to define alpha-equivalence. However, we need simple typing rules, see Fig. 1. Going forward, we only consider well-typed terms. The type of a term c corresponds to the number of open wires on each side of the diagram of c . A simple induction confirms uniqueness of types: if $c : m \rightarrow n$ and $c : m' \rightarrow n'$, then $m = m'$ and $n = n'$. A circuit with no input (*resp.* output) wires has type $0 \rightarrow n$ (*resp.* $m \rightarrow 0$).

Like in traditional Boolean circuits, our terms are intended to represent networks of logical gates connected by wires carrying Boolean values. We will assign a formal semantics in Section 2.2, but give some intuition for their interpretation now. The purely Boolean fragment of the syntax (BoolCirc) includes standard gates for logical conjunction, $\text{---} \text{---}$, which outputs a 1 only when its two inputs are set to 1; logical negation, $\text{---} \text{---}$, which negates its input; a broadcasting gate, $\bullet \text{---} \text{---}$, that copies its input to two output wires; and a terminating wire, $\bullet$, that discards its input. Beyond Boolean circuits, there are two additions. Firstly, CausCirc introduces $\text{---} \text{---}$, which is the only probabilistic component of our syntax, emitting 1 with probability p and 0 with probability $1 - p$. Note that $\text{---} \text{---}$ has *no input* and that the parameter p is simply a label which is not accessible to the rest of the circuit. Thus we have a generator for each $p \in [0, 1]$. Finally, ProbCirc adds a component for explicit conditioning, $\text{---} \text{---}$, which constrains its two input wires to carry equal values and outputs this value. A similar

$$\begin{array}{c} m \\ \text{---} \end{array} \boxed{c} \begin{array}{c} n \\ \text{---} \end{array} = m \left\{ \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \boxed{c} \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \right\} n$$

Then, we represent $c; d$ as sequential composition, depicted horizontally from left to right (note that the types of the intermediate wires have to match), and $c \otimes d$ as parallel composition, depicted vertically from top to bottom:

$$\ell \text{---} \boxed{c; d} \text{---} n = \ell \text{---} \boxed{c} \text{---} m \text{---} \boxed{d} \text{---} n \quad m_1 + m_2 \text{---} \boxed{c_1 \otimes c_2} \text{---} n_1 + n_2 = \frac{m_1}{m_2} \boxed{c_1} \frac{n_1}{n_2}$$

We omit the labels on the wires for readability when they can be inferred unambiguously from the context.

Intuitively, our string diagrams can be formed much like conventional circuits, by wiring the generators of (1)-(3) in sequence or in parallel, crossing wires (with $\times$) and making wires as long as we want (with ---). As string diagrams, the axioms of SMCs now become near-tautologies:

If we think of the dotted frames as two-dimensional brackets, these identities tell us that the specific bracketing of a term does not matter. This is precisely the advantage of working with string diagrams, rather than terms: they free us from some of the bureaucracy of terms, allowing us to focus on the more structural aspects of our syntax, *i.e.* on how the different components making up a term are wired together. This means in particular that wire crossings obey laws that are topologically obvious, as shown in (4).

Example 2.2. For the OR-gate and multiplexers defined in Example 2.1, we have

Even though we have not defined a formal semantics yet, their interpretation is clear from their depiction as circuits: the first will behave like a standard logical OR-gate, which outputs the disjunction of its two inputs, and the second like a multiplexer or *if-then-else*-gate, which outputs either its second input if the first is set to 1 or the third input if the first is set to 0. We will call the first input the *guard*, and the next two, the *then*- and *else-branch*, respectively.

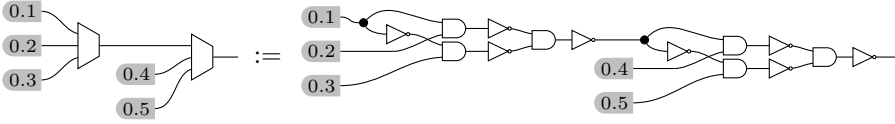
We also define generalised gates for n wires, as syntactic sugar, by induction, for AND-gates and copying nodes:

$$\begin{array}{c}
\frac{}{\Gamma, x : \tau, \Delta \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} \, e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd} \, e : \tau_2} \\
\\
\frac{\Gamma \vdash g : \mathbb{B} \quad \Gamma \vdash e_0 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathbf{if} \, g \, \mathbf{then} \, e_1 \, \mathbf{else} \, e_0 : \tau} \quad \frac{}{\Gamma \vdash \mathbf{flip} \, p : \mathbb{B}} \quad \frac{}{\Gamma \vdash \mathbf{true} : \mathbb{B}} \\
\\
\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma, x : \tau_0 \vdash e_1 : \tau}{\Gamma \vdash \mathbf{let} \, x = e_0 \, \mathbf{in} \, e_1 : \tau} \quad \frac{}{\Gamma, x : \tau, y : \tau, \Delta \vdash (x = y) : \tau}
\end{array}$$

Fig. 2: Typing rules for a simple language for discrete probabilistic programming. Metavariables x, y range over variable names, and p over reals in $[0, 1]$.

with base cases the corresponding generators; we will use generalised gates for $\rightarrow$, $\neg$, $\wedge$, $\vee$, etc. defined in a similar way.

Example 2.3. Consider a simple setting where we flip a coin x with a 0.1 chance of landing heads. If x is heads, we flip a new coin y_1 with a 0.2 chance of heads; otherwise, we flip a third coin y_2 with a 0.3 chance of heads. Finally, based on the outcome of y_i (for either $i = 1$ or $i = 2$), we choose between two more coins z_1 or z_2 : if y_i is heads, z_1 has a 0.4 chance of heads; otherwise, we flip z_2 , which is a fair coin. Now, imagine that we are interested in the probability of the last coin flip landing on heads. This is what the following causal circuit models:



Comparing with a conventional PL syntax. We now explain the relationship of our circuits with the more conventional syntax of the probabilistic programming language used in the introduction. Fig. 2 gives the syntax and typing rules of this simple (first-order) functional PPL: the judgment $\Gamma \vdash e : \tau$ denotes a well-typed partial program e in context Γ , which is a list of pairs $x : \tau$ of free variables with their types, and types are tuples of Booleans. This syntax is closely related to that of *Dice* [22] and of the *CD-calculus* [36, Section 6.3], domain-specific languages for discrete probabilistic programming.

At the level of the purely Boolean fragments, these programming languages and our circuits can express the same maps, though they use **if-then-else** as primitive, rather than conjunction and negation. Indeed both are well known to constitute universal sets of gates for Boolean functions. Instead of wires, a standard syntax uses *variables* to encode circuits inputs and **let**-expressions to encode circuit composition. The latter are not needed to represent plain Boolean functions—the algebraic syntax of Boolean algebra suffices for this purpose [5].

The first probabilistic primitive is **flip** p , which represents a coin flip with probability $p \in [0, 1]$ of landing on **true**, like our own $\mathbf{p}$. Note that simply adding **flip** p to the syntax of Boolean algebra would give a syntax that is

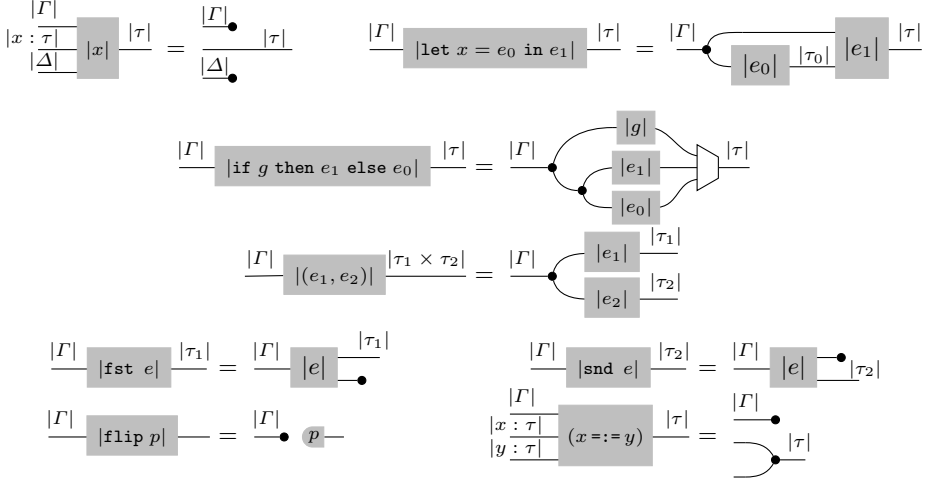


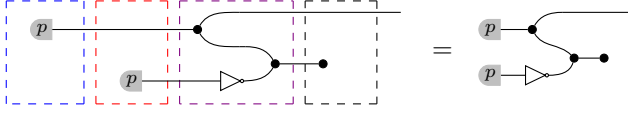
Fig. 3: Diagrammatic translation of programs to circuits. The mapping $|\cdot|$ is defined inductively: on types by $|\tau_1 \times \tau_2| = |\tau_1| + |\tau_2|$ with base case $|\mathbb{B}| = 1$, on contexts by $|Γ, x : \tau| = |Γ| + |\tau|$ with base case $|\emptyset| = 0$, and on (open) terms by the rules above.

insufficient to reuse (or ignore) the outcome of a `flip` p . Indeed there would be no way of binding its outcome to some variable for multiple use, contrary to our diagrammatic syntax, which can broadcast the output of p — to multiple sub-circuits using $\text{---}\bullet\text{---}$ (or discard it with $\text{---}\bullet$). This justifies the need for `let`-expressions: to mimic the non-linear use of probabilistic outcomes. Concretely, `let` $x = e_0$ `in` e_1 binds some term e_0 to the variable x for its use in e_1 .

The second probabilistic primitive is $(x := y)$, interpreted as evidence that the variables x and y are equal; it has the effect of conditioning the posterior distribution the program represents on the events for which these two variables agree. This construct appears in the work of Stein and Staton on exact conditioning [36]. Our circuit $\text{---}\text{---}\text{---}$ plays a similar role constraining its two inputs to have equal value. Some PPL, like Dice, use `observe` x instead to express conditioning on observations for which x is true [22]. This is equivalent to our choice, since we can simply encode it by setting one side of the $(x := y)$ to `true`. In the other direction, we can encode $(x := y)$ as `observe(not` x `xor` $y)$.

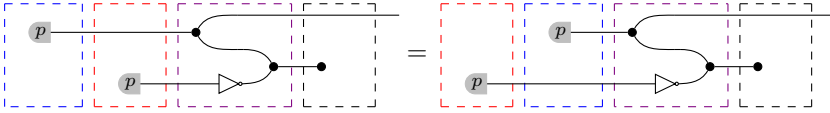
Based on this discussion, we can devise a more formal translation of our symbolic PPL into circuits. A (partial) program $\Gamma \vdash e : \tau$ will now be translated into a circuit $|e| : |\Gamma| \rightarrow |\tau|$ where $|\Gamma|$ is simply the length of the context Γ and, similarly, $|\tau|$ is the length of the corresponding tuple of Booleans. In other words, a partial program with m free variables and whose main expression has type $\mathbb{B}^n$ represents a circuit with m input wires and n output wires. A complete inductive translation on the typing rules is given in Fig. 3. Note that it uses the multiplexer and n -ary gates defined in the main text as syntactic sugar.

Example 2.4. The von Neumann’s trick program from Section 1 can be translated into the following circuit, where each dashed box corresponds to a line:



Intuitively, it is clear that this circuit flips two coins with the same parameter p as in the previous circuit, negates the outcome of one of them, conditions on the two tosses being equal, and asks for the probability that the first flip gave true.

The axioms of SMCs capture non-trivial program transformations: *e.g.*, the first two **let**-expressions can be swapped without changing the diagram³:



This example highlights one advantage of string diagrams—the topological transformations of the two-dimensional representation absorb some, but not all, of the burden of reasoning about program equivalence.

2.2 Semantics

Before establishing the formal semantics of our circuits, we recall some necessary background. In this paper, ‘subdistribution’ will always refer to a *finitely-supported discrete probability subdistribution*, *i.e.*, a map $\varphi : X \rightarrow [0, 1]$ where X is finite and such that $\sum_{x \in X} \varphi(x) \leq 1$. A *distribution* is then a subdistribution φ for which $\sum_{x \in X} \varphi(x) = 1$. We will write subdistributions as formal sums $\sum_x \varphi(x) |x\rangle$, omitting the elements of X for which $\varphi(x) = 0$, and call $\mathcal{D}_{\leq 1}X$ (resp. $\mathcal{D}X$) the set of all subdistributions (resp. distributions) over X .

The symmetric monoidal category of substochastic maps. Our circuits do not map inputs to outputs in a deterministic way. Instead, for each input, they specify a subdistribution over their outputs. Thus, we will interpret them as *substochastic maps* $f : X \rightarrow \mathcal{D}_{\leq 1}Y$, which we write as $f : X \rightharpoonup Y$. Currying the Y component, we can think of f as a map $f(-|-) : Y \times X \rightarrow [0, 1]$ or, equivalently, as a matrix whose columns sum to at most 1. We will switch freely between the two perspectives. As with subdistributions, a substochastic map is *stochastic* when $\sum_y f(y|x) = 1$ for all $x \in X$ (or when its columns sum to 1). Note that any map $f : X \rightarrow Y$ can be promoted to a substochastic map $X \rightharpoonup Y$ given by $x \mapsto |f(x)\rangle$ for all $x \in X$.

We can also interpret $f : X \rightharpoonup Y$ as specifying a *conditional* subdistribution: $f(-|x)$ gives a subdistribution over Y for each $x \in X$. The composition $f ; g$ of

³ A fact also known as the commutativity of probabilistic effects in PL parlance.

two substochastic maps $f : X \multimap Y, g : Y \multimap Z$ is given by summing over the intermediate variable as follows:

$$(f ; g)(z|x) = \sum_{y \in Y} g(z|y)f(y|x)$$

Note that, if we think of f and g as substochastic matrices, this is the formula for their product in the usual sense. The identity $\text{id}_X : X \multimap X$ is simply the Dirac delta $\delta_x : x \mapsto |x\rangle$ or, equivalently, the identity matrix. Substochastic maps with these operations define a category, which we call **FinSubStoch** [26]. Moreover, since stochastic maps are stable under composition, they form a subcategory of **FinSubStoch**, which we call **FinStoch** [13].

For two subdistributions $\varphi \in \mathcal{D}_{\leq 1}X$ and $\rho \in \mathcal{D}_{\leq 1}Y$, we can form the product subdistribution $\varphi \otimes \rho \in \mathcal{D}_{\leq 1}(X \times Y)$, given by $(\varphi \otimes \rho)(x, y) = \varphi(x) \cdot \rho(y)$. We will write $|xy\rangle = |x\rangle \otimes |y\rangle$ so that $(\sum_x \varphi(x) |x\rangle) \otimes (\sum_y \rho(y) |y\rangle) = \sum_{x,y} \varphi(x) \rho(y) |xy\rangle$. The same operation can be extended to conditional distributions, that is, to substochastic maps $f_1 : X_1 \multimap Y_1$ and $f_2 : X_2 \multimap Y_2$, giving $(f_1 \otimes f_2)(x_1, x_2) = f_1(x_1) \otimes f_2(x_2)$. This makes **FinSubStoch** into a *symmetric monoidal* category, with the Cartesian product of sets as monoidal product, the singleton set $1 = \{\bullet\}$ as unit, and the symmetry $\sigma_Y^X : X \times Y \multimap Y \times X$ given by $\sigma_Y^X(x, y) = |yx\rangle$. Stochastic maps out of the unit 1 , e.g. $1 \multimap X$, correspond precisely to distributions over X . Since the product of two distributions is still a distribution, **FinStoch** is a symmetric monoidal subcategory of **FinSubStoch**.

Remark 2.5. There is another way of making **FinStoch** into a SMC, with the monoidal product given by disjoint sum of sets on objects and direct sum of matrices on morphisms. This SMC has already been given a complete axiomatisation in the work of Fritz [12], which is itself a categorical reformulation of the much earlier work of Stone [38] on barycentric algebras. No axiomatisation exists for the monoidal product we consider here.

Copying and deleting. Note that $\mathcal{D}(1) = 1$ and therefore there is only one stochastic map $\epsilon_X : X \multimap 1$ for any X , given by $\epsilon_X(x) = |\bullet\rangle$. Moreover, there is a canonical diagonal substochastic map inherited from the Cartesian product of sets: $\Delta_X : X \multimap X \times X$ given by $\Delta_X(x) = |xx\rangle := |x\rangle \otimes |x\rangle$. It is important to note that $f ; \Delta_Y \neq \Delta_X ; (f \times f)$ in general—we say that arbitrary substochastic maps cannot be copied. Intuitively, this makes sense: if f represents some probabilistic process (such as flipping a coin), running f once and reusing its outcome in two different places is different from running f twice (flipping two coins).

Similarly, not every substochastic map $f : X \multimap Y$ satisfies $f ; \epsilon_Y = \epsilon_X$. Those that do are *stochastic*, since $(f ; \epsilon_Y)(x) = \sum_{y \in Y} f(y|x) = 1 = \epsilon_X(x)$. Such maps are often called *discardable* or *causal* [13, 6], which is why we refer to conditioning-free circuits as ‘causal circuits’. We will later see that they are precisely those circuits whose semantics give a bona-fide stochastic map.

Furthermore, using ϵ , we can recover the usual notion of *marginal* from probability theory. Indeed, post-composing a stochastic map $f : 1 \multimap X \times Y$ with ϵ_Y

involves summing over all $y \in Y$ as follows: $f; (\text{id}_X \times \epsilon_Y)(x) = \sum_{y \in Y} f(x, y)$. If we think of f as a joint distribution over variables taking values in X and Y , the stochastic map $f; (\text{id}_X \times \epsilon_Y)$ thus corresponds precisely to marginalising over Y to obtain a distribution over X only.

Conditionals and disintegration. Similarly, we can translate the notion of *conditioning* in the language of **FinStoch**. A stochastic map $f : X \rightarrow Y$ can be thought of as a conditional distribution $f(-|x)$ for each $x \in X$ and facts about conditionals have their counterpart in **FinStoch**. In particular, given a stochastic map $f : 1 \rightarrow X \times Y$, there exists $f_{|X} : X \rightarrow Y$ such that $f = f; (\text{id}_X \times \epsilon_Y); \Delta_X; (\text{id}_X \times f_{|X})$. This is a direct translation of the *disintegration* of a joint distribution $\mathbb{P}(x, y) = \mathbb{P}(x)\mathbb{P}(y|x)$ into the product of a marginal and a conditional distribution. In the language of **FinStoch**, $f; (\text{id}_X \times \epsilon_Y)$ corresponds to the marginal $\mathbb{P}(x)$, and $f_{|X}$ to the conditional $\mathbb{P}(y|x)$.

It is known that such disintegrations are *not* unique in general. Indeed, when the marginal distribution $\mathbb{P}(x)$ is not fully-supported ($\mathbb{P}(x) = 0$ for some $x \in X$), then $\mathbb{P}(y|x)$ can be arbitrary. It is clear that any two disintegrations of the same distribution (for the same order of the variables) can have conditionals that differ only on a set of measure zero for the corresponding marginal [6, Section 6].

Proposition 2.6 (Almost-sure uniqueness of disintegrations). *Given two disintegrations $(g, f_{|X})$ and $(g', f'_{|X})$ of the stochastic map $f : A \rightarrow X \times Y$, then $g = g' = f; (\text{id}_X \times \epsilon_Y)$ and $f_{|X}(a, x) = f'_{|X}(a, x)$ for all $a \in A$ and $x \in X$ such that $g(x|a) = g'(x|a) \neq 0$.*

Interpreting causal circuits. We are now ready to define the semantics of causal circuits. To do so, we will use the standard Boolean operations of conjunction (written ‘ $\wedge$ ’) and negation (written ‘ $\neg$ ’).

Definition 2.7 (Semantics of causal circuits). *Let $\llbracket \cdot \rrbracket$ be the mapping defined inductively on **CausCirc** by*

$$\begin{aligned} \llbracket \text{---} \circ \text{---} \rrbracket (x) &= |xx\rangle & \llbracket \text{---} \bullet \rrbracket (x) &= |\bullet\rangle \\ \llbracket \text{---} \triangleright \rrbracket (x) &= |\neg x\rangle & \llbracket \text{---} \text{p} \text{---} \rrbracket (\bullet) &= p|1\rangle + (1-p)|0\rangle \\ \llbracket \text{---} \rrbracket (x) &= |x\rangle & \llbracket \text{---} \times \rrbracket (x, y) &= |yx\rangle & \llbracket \text{---} \text{---} \rrbracket (x_1, x_2) &= |x_1 \wedge x_2\rangle \\ \llbracket \text{---} \text{c} \text{---} \text{d} \text{---} \rrbracket (z|x) &= \sum_{y \in Y} \llbracket \text{---} \text{d} \text{---} \rrbracket (z|y) \cdot \llbracket \text{---} \text{c} \text{---} \rrbracket (y|x) \\ \llbracket \text{---} \text{c}_1 \text{---} \text{c}_2 \text{---} \rrbracket^{n_1}_{n_2} (y_1, y_2|x_1, x_2) &= \llbracket \text{---} \text{c}_1 \text{---} \rrbracket^{n_1}_{n_1} (y_1|x_1) \cdot \llbracket \text{---} \text{c}_2 \text{---} \rrbracket^{n_2}_{n_2} (y_2|x_2) \end{aligned}$$

Note that the first line of Definition 2.7 is the usual semantics of these circuit gates, given in terms of Boolean operations, lifted to **FinStoch**. The second line defines the semantics of p to be a Bernoulli distribution with parameter p . The next line forces $\llbracket \cdot \rrbracket$ to map plain wires and wire crossings to identities and symmetries in **FinStoch**. As we see, the image of every primitive lands in **FinStoch**,

i.e., is a (normalised) distribution. Finally, the last two lines guarantee that our interpretation is *compositional* : the semantics of two circuits composed in sequence is their composition in **FinStoch** and the semantics of two circuits in parallel is their monoidal product. Thus, a circuit $c : m \rightarrow n$, with m input wires and n output wires, is interpreted as a stochastic map $\llbracket c \rrbracket : \mathbb{B}^m \rightarrow \mathbb{B}^n$, *i.e.*, as a map $\mathbb{B}^m \rightarrow \mathcal{D}(\mathbb{B}^n)$. As a result, this interpretation does define a symmetric monoidal functor from syntax to semantics.

Proposition 2.8. *The map $\llbracket \cdot \rrbracket$ is a symmetric monoidal functor from **CausCirc** to **FinStoch**.*

Semantics of conditioning. Famously, explicit conditioning complicates the semantics of probabilistic programs. Here, we want the generator $\bowtie$ of our syntax to constrain both its inputs to have the same value. Hence, its interpretation should return the Dirac distribution $|x\rangle$ whenever its two inputs are both equal to x , but what should its output subdistribution be when its two inputs disagree? Following standard practice [7,22,23], we will assign probability 0 to such failed conditions. This suggests extending $\llbracket \cdot \rrbracket$ to all of **ProbCirc** with

$$\llbracket \bowtie \rrbracket (y|x_1, x_2) = \begin{cases} 1 & \text{if } y = x_1 = x_2, \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Note that this does define an unnormalised *subdistribution*. As a result, conditioning forces us to leave **FinStoch**, the subcategory of (normalised) distributions.

However, it is always possible to rescale a non-zero subdistribution to obtain a distribution, as has been proposed for the semantics of many PPLs [7,22,11,29]. And indeed, for the purpose of inference, any two closed probabilistic programs that represent the same distribution, should be considered equal. Equivalently, we would like to identify any two circuits $0 \rightarrow n$ whose semantics give proportional subdistributions. While there are other interpretations of conditioning in probabilistic programming [29], this is the perspective we adopt here. Let us make this idea more precise, following the work of Stein and Staton [36].

For any two $\phi, \rho \in \mathcal{D}_{\leq 1}X$, we write $\phi \propto \rho$ if there exists some real number $\lambda > 0$ such that $\phi(x) = \lambda \cdot \rho(x)$. This defines an equivalence relation on $\mathcal{D}_{\leq 1}X$. Moreover, since any non-zero subdistribution can always be normalised, the equivalence classes of $\propto$ are in one-to-one correspondence with distributions over X , plus the uniformly zero subdistribution $\perp_X$, *i.e.*, $\mathcal{D}_{\leq 1}X / \propto \cong \mathcal{D}X + \{\perp_X\}$. In this sense, conditioning just adds the possibility of failure, denoted by $\perp_X$, to the standard distributional semantics given by $\llbracket \cdot \rrbracket$.

Thus, our intended semantics should identify circuits of type $0 \rightarrow n$ whose subdistributional semantics differ only by a constant (non-zero) multiplicative factor. To interpret programs with free variables/circuits with input wires, we need to generalise the idea to all substochastic maps. Moreover, we need to do so in a *compositional* way: this means that the semantics for circuits $0 \rightarrow n$ should extend to a semantics for arbitrary circuits $m \rightarrow n$ in a way which is compatible with sequential and parallel composition.

It turns out that there is only one consistent way to do so. First, note that if we identify subdistributions up to a scalar factor, we have to identify all subdistributions over the set $1 = \{\bullet\}$, and therefore all circuits of type $0 \rightarrow 0$. But subdistributions over 1 are precisely the substochastic maps $1 \rightarrow 1$, and these are in one-to-one correspondence with the non-negative reals. This tells us exactly how to extend our equivalence relation $\propto$ over subdistributions to substochastic maps: consider $c : 1 \rightarrow 1$ and $f : X \rightarrow Y$; then c is equivalent to id_1 , the identity map $1 \rightarrow 1$, which implies that $c \times f$ should be equivalent to $\text{id}_1 \times f = f$. Since c was arbitrary, we have that any two substochastic maps that differ by a global nonzero multiplicative factor should be equivalent.

Definition 2.9. *Given two substochastic maps $f, g : X \rightarrow Y$, we write $f \propto g$ if there exists a real number $\lambda > 0$ such that $f(y|x) = \lambda \cdot g(y|x)$ for all $x \in X$ and $y \in Y$. We write $[f]$ for the equivalence class of $f : X \rightarrow Y$.*

Note that λ has to be same scalar factor for all inputs in X (see paragraph preceding Definition 2.9). With this assumption, composition and product of stochastic maps are congruences for $\propto$, so that stochastic maps up to $\propto$ form a symmetric monoidal category, which we call **FinProjStoch**, with much of the same structure as **FinStoch** [36, Theorem 6.4]. **FinProjStoch** will be our target semantics for **ProbCirc**, the syntax of all circuits, including those that contain explicit conditioning via $\rhd$ generators.

Definition 2.10 (Semantics of all circuits). *For each generator g of **CausCirc**, let $\llbracket g \rrbracket_\propto$ be the equivalence class of $\llbracket g \rrbracket$. In addition, let $\llbracket \rhd \rrbracket_\propto$ be the equivalence class of the subdistribution defined in (5). Finally, sequential and parallel composition are interpreted as in Definition 2.7.*

Proposition 2.11. *The map $\llbracket \cdot \rrbracket_\propto$ defines a symmetric monoidal functor from **ProbCirc** to **FinProjStoch**.*

Props and symmetric monoidal theories. Following standard practice, we can see our circuits as morphisms of a product and permutation category, or *prop*. Formally, a prop is a strict SMC with the natural numbers as the set of objects and addition as the monoidal product. In a prop, all objects are thus monoidal products of a single generating object, *viz.* 1.

Given a set of generating morphisms Σ , we can form the free prop P_Σ as explained in Section 2.1: by quotienting Σ -terms by the laws of SMC. For us here, **ProbCirc** is the free prop over the set of generators in (1)-(3) and **CausCirc** the free prop over those in (1)-(2).

Now that we have a formal syntax and semantics, our aim is to reason about semantic equivalence of circuits purely equationally. As we saw, since the syntax of circuits is a prop and its interpretation $\llbracket \cdot \rrbracket$ is a symmetric monoidal functor (Proposition 2.11) into another SMC, the laws of SMCs already capture some semantic equivalences between circuits, but they are not sufficient to derive all valid equivalences. The main aim of this paper is to add enough axioms to obtain a completeness result.

Given a set E consisting of equations between Σ -terms, we write $=_E$ for the smallest congruence with respect to the two compositions $;$ and $\otimes$ containing E . We call the elements of E axioms and the pair (Σ, E) a *symmetric monoidal theory* (or simply, *theory*). Details on the existence and construction of free props on a given theory can be found in [2, Appendix B] or [40]. We say that a theory is *sound* if $c =_E d$ implies $\llbracket c \rrbracket = \llbracket d \rrbracket$ and *complete* when the reverse implication holds. A sound and complete theory is also called an *axiomatisation*. When moreover, for every morphism f of the target semantics there exists a morphism c in P_Σ , the syntax, such that $\llbracket c \rrbracket = f$, we say that a sound and complete theory is a *presentation* of the image of $\llbracket \cdot \rrbracket$.

In what follows we give a sound and complete theory for equivalence of causal circuits (Section 3). Equivalently, this will provide a presentation of $\text{FinStoch}_{\mathbb{B}}$, the SMC of stochastic maps between sets that are powers of $\mathbb{B}$. For our second main result, we extend this theory to axiomatise the full language of probabilistic circuits, including explicit conditioning (Section 4).

3 An Axiomatisation of Causal Circuits

We now focus on causal circuits, the axiomatisation of which is of independent interest, since it gives a presentation of a monoidal subcategory of FinStoch .

3.1 Equational Theory

We consider CausCirc terms quotiented by the set of equations in Fig. 4 and the laws of symmetric monoidal categories (4). Note that any axiom which uses parameters $p, q, \dots$ implicitly quantifies over $[0, 1]$, unless stated otherwise.

A few comments are in order, to clarify the meaning of the axioms: in the first block, the **A**-axioms define a *cocommutative comonoid* consisting of a *comultiplication* $\dashv\!\!\!\dashv : 1 \rightarrow 2$ and a *counit* $\dashv\!\!\!\dashv : 1 \rightarrow 0$, together with equations guaranteeing that all different ways of copying a value are equal, that copying followed by discarding is the same as doing nothing, and that two copies are identical, so can be exchanged; in the second block, axioms **B1-B3** define a *commutative monoid* consisting of two generators in our signature, a *multiplication*, given by the Boolean AND-gate $\curlywedge : 2 \rightarrow 1$, and a *unit*, the Boolean constant **true**, represented by $\mathbf{1} : 0 \rightarrow 1$, with their associated axioms: $\curlywedge$ is associative, commutative and its unit is $\mathbf{1}$; in the third block, axioms from Boolean algebra are given, stating the *involution* of negation, the *idempotence* of conjunction, the usual *complementation* axiom and the *distributivity* of conjunction over disjunction; in the fourth block, the axioms allow us to copy (**C**) arbitrary Boolean operations using $\dashv\!\!\!\dashv$. In other words, conjunction, negation, and the two Boolean constants **true** (1) and **false** (0) distribute over $\dashv\!\!\!\dashv$. Semantically, this characterises them as *deterministic* maps without any probabilistic behaviour; in the fifth block, the **D**-axioms allow us to discard (**D**) the result of Boolean operations, using $\dashv\!\!\!\dashv$. Note that axiom **D3** applies more generally to $\mathbf{p} : 0 \rightarrow 1$ for any $p \in [0, 1]$ and encodes the normalisation of the distribution

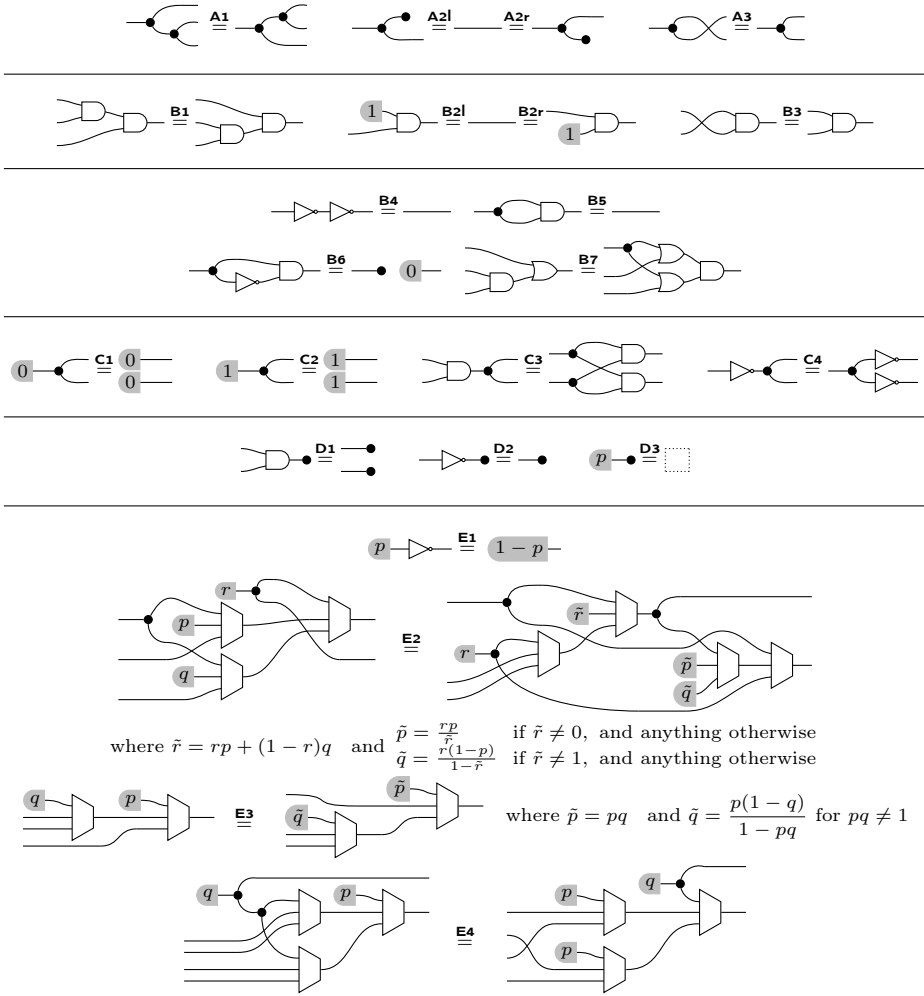


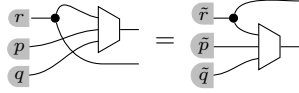
Fig. 4: Axioms for causal circuits.

$\llbracket p \rrbracket = p |1\rangle + (1 - p) |0\rangle$, i.e. the fact that $p + (1 - p) = 1$. Semantically, this characterises these as *causal* operations.

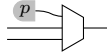
Note that the **A-D** blocks allow us to obtain a diagrammatic form of substitution for the purely Boolean fragment of the theory: it guarantees that any *purely Boolean* expression can be copied (**C**) and discarded (**D**). In addition, they give us algebraic structures that occur in similar diagrammatic calculi: $\text{---}\text{---}$, $\text{---}\text{---}$ with $\text{---}\text{---}$, $\text{---}\text{---}$ together form a (co)commutative *bimonoid*.

Finally, the last block contains all equations concerning the probabilistic behaviour of **CausCirc** terms. Axiom **E1** is clear: applying a negation after a coin flip with bias p is the same as exchanging the probabilities of the corresponding

distribution, *i.e.*, $p|0\rangle + (1-p)|1\rangle = (1-p)|\neg 0\rangle + p|\neg 1\rangle$. This implies in particular that $|0\rangle$ is the negation of $|1\rangle$, which is a simple Boolean algebra identity. Axiom **E2** generalises a simpler equality, which is the diagrammatic counterpart of two ways of disintegrating a joint distribution of two variables, as $\mathbb{P}(y_0|y_1)\mathbb{P}(y_1) = \mathbb{P}(y_0)\mathbb{P}(y_1|y_0)$:


(6)

E2 further conditions this identity on the value of some input variable (the top left wire). This more general form will be needed to iteratively disintegrate joint distributions over more than two variables, in the proof of completeness. Note that by varying the values of p, q, r in this last diagram, the semantics of the corresponding circuit ranges over all distributions on $\mathbb{B}^2$. Axiom **E3** can be understood as form of associativity for convex sums. Indeed, circuits of the form



for some $p \in [0, 1]$, are interpreted as the distribution $p|x_0\rangle + (1-p)|x_1\rangle$, conditional on inputs x_0 and x_1 . That is, circuits of this form simply take the convex sum of their two inputs. This is a binary operation, which is associative *up to reweighing* as axiom **E3** states. Similar axioms appear in previous work that contain convex sum (aka probabilistic choice) as a primitive [38,12,33]. A difference with our work is that convex sum is a derived operation for us. This will allow us to derive laws commonly taken as axioms in related work, most notably the *distributivity* of **if-then-else** over probabilistic choice [33, Section 5]. Axiom **E4** allows us to break redundant correlation between the guard of two **if-then-else** gates: notice that there is one fewer $\neg$ node on the right than on the left. This ability will prove crucial in our proof of completeness, helping us to rewrite complicated circuits into simple trees of convex sums.

3.2 Soundness and Completeness

Using our equational theory, every syntactically provable equality holds in the semantics. The proof of the theorem below is straightforward, as it suffices to verify that each axiom in Fig. 4 is a semantic equality.

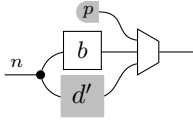
Theorem 3.1 (Soundness). *For circuits $c, d : m \rightarrow n$, if $c = d$ then $\llbracket c \rrbracket = \llbracket d \rrbracket$.*

To prove completeness, we use a normal form argument, a common strategy in completeness proofs for diagrammatic calculi [31,20]. The idea is that normal forms provide a unique syntactic representative for each semantic object in the image of the interpretation functor $\llbracket \cdot \rrbracket$, thereby guaranteeing that, if two circuits denote the same distribution, they will be equal to the same normal form. The proof then relies on a normalisation procedure: an explicit algorithm which rewrites any given circuit into normal form, using only the proposed axioms.

Our normalisation argument proceeds in two high-level steps: 1) we give a procedure to normalise $n \rightarrow 1$ causal circuits; 2) we then reduce the case of general $m \rightarrow n$ causal circuits to multiple applications of the procedure for the $n \rightarrow 1$ case, by syntactically disintegrating a given circuit.

Single-output circuits. The normalisation procedure for $n \rightarrow 1$ circuits begins by rewriting a given circuit into *pre-normal form*. This pre-normal form intuitively represents a diagram as a convex combination of Boolean expressions.⁴

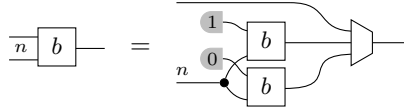
Definition 3.2 (Pre-normal form). *A circuit $d : n \rightarrow 1$ is in pre-normal form if it is in the form defined inductively below:*



where $b : n \rightarrow 1$ is a Boolean circuit and $d' : n \rightarrow 1$ is itself in pre-normal form or is a Boolean circuit (base case).

The core of rewriting a circuit into pre-normal form is to move all occurrences of the probabilistic generators $\textcircled{p}$ to the guard of some **if-then-else** gate, and then to eliminate correlations between different branching gates sharing the same $\textcircled{p}$ (via $\text{---}\textcircled{\curvearrowright}$ nodes). We do the first step through repeated usage of *Shannon expansion*, a well-known consequence of the Boolean algebra axioms (**A-D**).

Lemma 3.3 (Shannon expansion). *For every Boolean circuit $b : 1 + n \rightarrow 1$, we have*



While Shannon expansion can introduce further correlations between the guards of different **if-then-else** gates, we remove these with axiom **E4**, to reach our pre-normal form. This process itself might leave a trail of convex sums, which we need to re-associate in the correct way using axiom **E3** to obtain a circuit in pre-normal form.

Lemma 3.4 (Pre-normalisation). *Every $n \rightarrow 1$ causal circuit is equal to one in pre-normal form.*

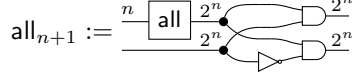
Our pre-normal form is not unique: different circuits may be mapped to the same stochastic map by $\llbracket \cdot \rrbracket$. We thus have to establish a method for showing that any two semantically equivalent circuits in pre-normal form are equal. To do so, we define a suitable *normal form* for single-output circuits. Given some circuit $c : n \rightarrow 1$, its normal form will be a syntactic encoding of the probability

⁴ This can be seen as a generalisation of Birkhoff's theorem for doubly stochastic matrices [3] to the (singly-)stochastic case.

table of $\llbracket c \rrbracket : \mathbb{B}^n \rightarrow \mathbb{B}$: a disjunction of the Bernoulli distribution, in the form of a single p —, for every possible input. Thus, for any input $x \in \mathbb{B}^n$, we can read the probability of $\llbracket c \rrbracket(x)$ immediately from the normal form of c .

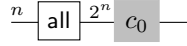
To define the normal form, we will need circuits $\text{all}_n : n \rightarrow 2^n$ which encode all possible 2^n bit vectors of length n , that is, all possible inputs on n wires.

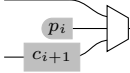
Definition 3.5. Let $\text{all}_n : n \rightarrow 2^n$ be a family of circuits defined by induction on $n \in \mathbb{N}$ with $\text{all}_0 := \text{1}$ — and



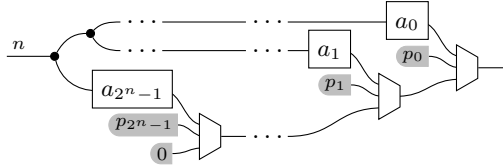
In other words, all_n demultiplexes a n -bit vector into a 2^n -bit vector where exactly one wire can be 1 at the same time, corresponding to the number encoded by the input.

Definition 3.6 (Normal Form for $n \rightarrow 1$ causal circuits). A circuit $n \rightarrow 1$ is in normal form when it is of the form



where $c_0 : 2^n \rightarrow 1$ is a circuit defined recursively as follows: $c_{2^n} := \text{0}$ — and $c_i :=$  for some $p_i \in [0, 1]$.

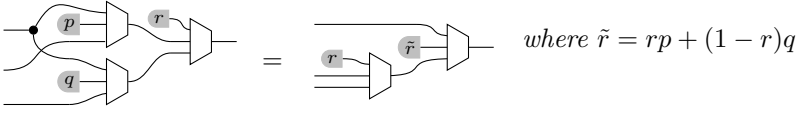
One way to visualise the normal form more easily is as follows:



where each $a_i : n \rightarrow 1$ corresponds to the circuit-encoding of the i -th bit vector of $\mathbb{B}^n$ in the order given by $\text{all}_n : n \rightarrow 2^n$. Since it is simply a syntactic encoding of the conditional probability table of $\llbracket c \rrbracket$ for a given $c : n \rightarrow 1$, this normal form is clearly unique.

It remains to show that any causal circuit of type $n \rightarrow 1$ is equal to one in normal form, by giving a procedure to rewrite it as such using the axioms of our theory. The key idea of this normalisation procedure is to start from the pre-normal form, and progressively aggregate all probabilities coming from the Boolean components of the convex sum that the pre-normal form represents. More specifically, for all possible bit-vector inputs $x \in \mathbb{B}^n$ for a circuit $\llbracket c \rrbracket : \mathbb{B}^n \rightarrow \mathbb{B}$, we need to take a convex sum of all the outputs coming from each of the Boolean components in the pre-normal form at x , weighted by the probability given by the guard of the corresponding **if-then-else** gate. This is done by exploiting a special case of axiom **E2**, obtained by discarding the second output on both the rhs and lhs of the axiom:

Lemma 3.7. *The following equality is derivable for all $r, p, q \in [0, 1]$:*

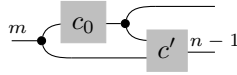


We can then establish our first completeness result:

Lemma 3.8 (Completeness for $n \rightarrow 1$ causal circuits). *Every causal circuit of type $n \rightarrow 1$ is equal to one in normal form.*

Multi-output circuits. For the completeness of $m \rightarrow n$ causal circuits, the main idea is to reduce the problem to the $n \rightarrow 1$ case by reproducing the disintegration of the corresponding distribution syntactically. The normal form is defined as a circuit disintegrated in the order of the output wires, from top to bottom.

Definition 3.9 (Normal form for $m \rightarrow n$ causal circuits). *A circuit $c : m \rightarrow n$ is in normal form when it is of the form*



where $c_0 : m \rightarrow 1$ in normal form in the sense of Definition 3.6 and $c' : m \rightarrow n - 1$ is in normal form, with the following additional requirement: if $\llbracket c_0 \rrbracket(b|x)$ has probability 0 for $(b, x) \in \mathbb{B} \times \mathbb{B}^m$, then $\llbracket c' \rrbracket(b, x) = |0^{n-1}\rangle$.

The last condition of the above definition is here to deal with the non-uniqueness of disintegrations. As we saw in §2.2, there are several possible disintegrations of a joint distribution $\mathbb{P}(x, y)$ into a marginal $\mathbb{P}(x)$ and a conditional $\mathbb{P}(y|x)$ if the marginal does not have full support—in this case, the value of $\mathbb{P}(y|x)$ for x such that $\mathbb{P}(x) = 0$ can be arbitrary. Because we want our normal forms to represent a given stochastic map uniquely, we need to fix a syntactic convention to handle such cases. Our chosen convention is to place all the probability mass on one arbitrary value, namely 0, so that $\mathbb{P}(y|x) = |0\rangle$.

Proposition 3.10 (Uniqueness of normal forms). *Any two circuits $c, d : m \rightarrow n$ in normal form and such that $\llbracket c \rrbracket = \llbracket d \rrbracket$, are equal.*

We rewrite a given circuit $c : m \rightarrow n + 1$ as the composition of an $m \rightarrow 1$ circuit whose semantics is that of the marginal distribution on the variable corresponding to the first output wire, and a circuit $m + 1 \rightarrow n$ whose semantics is that of the conditional distribution of the variable corresponding to the second wire, conditional on the first. Semantically, this amounts to disintegrating a joint distribution $\mathbb{P}(x_0, \dots, x_n)$ into the product $\mathbb{P}(x_0)\mathbb{P}(x_1, \dots, x_n|x_0)$, by conditioning on the first variable. Once again, axiom **E2** is the engine of the proof: its lhs and rhs represent the two different ways of disintegrating a distribution on two output wires/variables (conditional on some inputs, allowing us to reproduce the

process by going through the normal form while preserving its structure). In this way, starting from a circuit $m \rightarrow n$, we can obtain n circuits of type $m + k \rightarrow 1$, with $0 \leq k \leq n - 1$ by repeatedly computing disintegrations by induction on the number of output wires. By exploiting $n \rightarrow 1$ completeness, we obtain the following theorem.

Theorem 3.11 (Completeness for $m \rightarrow n$ causal circuits). *Every causal circuit of type $m \rightarrow n$ is equal to one in normal form. Thus, for any two causal circuits $c, d : m \rightarrow n$, if $\llbracket c \rrbracket = \llbracket d \rrbracket$ then $c = d$.*

Finally, by characterising the image of $\llbracket \cdot \rrbracket : \text{CausCirc} \rightarrow \text{FinStoch}$, we can conclude that our theory *presents* $\text{FinStoch}_{\mathbb{B}}$, the full monoidal subcategory of FinStoch on objects that are powers of the two-element set, i.e., $\mathbb{B}^n$ for $n \in \mathbb{N}$.

Proposition 3.12. *The image of CausCirc under $\llbracket \cdot \rrbracket$ is $\text{FinStoch}_{\mathbb{B}}$: for every morphism f of $\text{FinStoch}_{\mathbb{B}}$, there exists a causal circuit c such that $\llbracket c \rrbracket = f$.*

Theorem 3.13. *The theory given by the generators in (1)-(2) and the equations of Fig. 4 is a presentation of $\text{FinStoch}_{\mathbb{B}}$.*

Remark 3.14 (Encoding distributions over arbitrary finite sets). A language that can model any distribution over binary variables is also able to encode arbitrary distributions over finite sets, e.g., using what is commonly known as a *one-hot encoding*: a standard way to encode a set X as its copy inside the larger $\mathbb{B}^X$. To define it, we will identify $\mathbb{B}^X$ with the powerset of X . Given a distribution φ over some finite set X , we can define a distribution $O(\varphi)$ over $\mathbb{B}^X$ by $O(\varphi)(x) = \varphi(x)$ for $x \in X$ and $O(\varphi)(U) = 0$ for non-singleton subsets U of X . On the syntax side, we encode a distribution over X by a circuit with $|X|$ wires, whose semantics is a distribution over $\mathbb{B}^{|X|}$. Thus, each element of X corresponds to a single (output) wire of the associated circuit, but different wires are not allowed to be set to **true** at the same time. One-hot encodings are ubiquitous in computer science. For example, the designers of the language Dice use it to define probabilistic programs over arbitrary finite sets of events [22].

The one-hot encoding can be extended straightforwardly to substochastic maps: for $f : X \rightarrowtail Y$, let $OH(f)$ be the substochastic map given by $OH(f)(x) = O(f(x))$ for $x \in X$ and $OH(f)(U)$ the zero-everywhere subdistribution if U is not a singleton subset. Thus, we can use our circuits and the associated equational theory to reason about the whole of FinStoch (over arbitrary finite sets rather than powers of the two-element set), with some caveats, which we now explain.

OH does not define a functor from FinSubStoch to itself. While this encoding preserves composition, it does not preserve identities or the monoidal product. Indeed $OH(\text{id}_X)$ is not the identity stochastic map over $\mathbb{B}^X$. By the definition above, we see that it only acts as the identity over the singletons, and is the zero subdistribution otherwise. While this is not the identity needed to define a functor, $OH(\text{id}_X)$ is an idempotent map. This suggests one way to fix the issue: OH is better thought of as a functor from FinStoch into its Karoubi envelope, whose objects are pairs (X, e) of a set X and an idempotent $e : X \rightarrowtail X$. The categorical aspects of one-hot encoding are interesting in their own right, but they are not our focus, so we will not pursue this conjecture formally here.

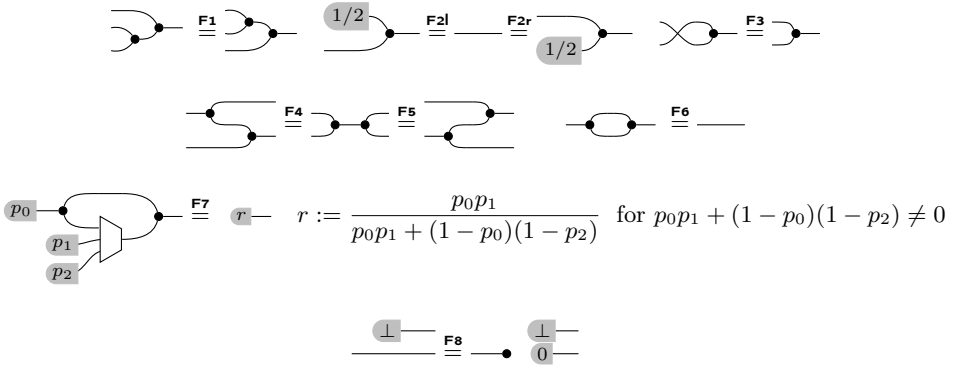


Fig. 5: Axioms for conditioning.

4 Adding conditioning

The power and usefulness of PPLs comes from their ability to specify distributions conditional on observations defined by the program. Our minimal PPL had a construct $(x ::= y)$ to condition on the value of x and y being equal, and our circuits have $\bigcirc$ for the same purpose. In this section, we extend the axiomatisation of Section 3 to all circuits, including those with explicit conditioning.

4.1 Equational theory

The extended equational theory consists of the axioms for causal circuits (Fig. 4) and those given in Fig. 5.

Axioms **F1-F3** make $\bigcirc$ and $\frac{1}{2}$ into a commutative monoid. The unitality axiom (**F2**) is only sound when we quotient subdistributions by a global multiplicative factor (see Definition 2.9). Had we given our semantics in terms of plain subdistributions (without quotient), these two equalities would not hold. Axioms **F4-F6** state that $\bigcirc$, $\bullet$, $\bigcirc$, $\frac{1}{2}$ form a *special Frobenius algebra*.

Axiom **F7** encodes the action of conditioning on a distribution over two variables. If we call x_1, x_2 these two variables we can see that the diagram on the right gives this distribution as the product of a Bernoulli with parameter p_0 for x_1 with the distribution of x_2 conditional on x_1 , given by two Bernoulli with parameters p_1 and p_2 . Clearly, x_1 and x_2 are both **true** with probability $p_0 p_1$ and both **false** with probability $(1-p_0)(1-p_2)$. Since $\bigcirc$ has the effect of constraining x_1 and x_2 to be equal, the result is a new Bernoulli distribution with parameter r , whose denominator is the normalisation factor $p_0 p_1 + (1-p_0)(1-p_2)$ and whose numerator is the probability that x_1 and x_2 are both **true**.

Axiom **F8** deals with failure, that is, with the special circuit $\perp := \frac{1}{0}$. It constrains 0 to be equal to 1, which is unsatisfiable. Any circuit that contains failure is semantically equal to the zero subdistribution, a fact which we will derive using axiom **F8** (Lemma 4.6).

Theorem 4.1 (Soundness). *For $c, d : m \rightarrow n$, if $c = d$ then $\llbracket c \rrbracket_\alpha = \llbracket d \rrbracket_\alpha$.*

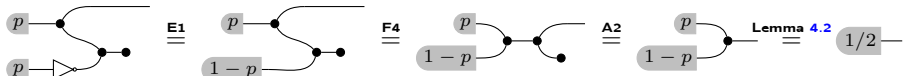
From these axioms, we can derive the action of $\curvearrowright$ on $\langle p \rangle$ generators: it multiplies the parameters and re-normalises the resulting subdistribution.

Lemma 4.2. *The following equality is derivable for all $p, q \in [0, 1]$ satisfying $pq + (1 - p)(1 - q) \neq 0$:*

$$\begin{array}{c} p \\ \text{---} \end{array} \begin{array}{c} \text{---} \\ \text{---} \end{array} \bullet \text{---} = \begin{array}{c} r \\ \text{---} \end{array} \quad \text{where } r := \frac{pq}{pq + (1-p)(1-q)}$$

We can now prove the correctness of the two examples from the introduction.

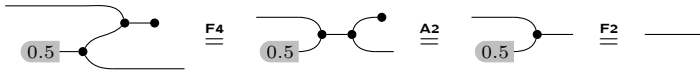
Example 4.3 (von Neumann's trick, correctness). Assume $p \in (0, 1)$. We have



since

$$\frac{p(1-p)}{p(1-p) + (1-p)(1-(1-p))} = \frac{p(1-p)}{p(1-p) + (1-p)p} = \frac{p(1-p)}{2p(1-p)} = \frac{1}{2}$$

Example 4.4. Using the axioms for Frobenius algebras, we can show the partial program from Section 1 (translated into the leftmost diagram below) is equivalent to the identity function on Boolean variables $\mathbf{x} : \mathbb{B} \vdash \mathbf{x} : \mathbb{B}$:



4.2 Completeness

We now show the completeness of the proposed equational theory.

First, we reduce the problem to the case of circuits without input wires, that is, circuits of type $0 \rightarrow n$. This is possible because the Frobenius algebra structure $(\text{---}\text{C}\text{---}, \text{---}\bullet, \text{---}\text{C}\text{---}, \text{---}\text{1/2---})$ with which we can bend wires using $\text{---}\text{1/2---}\text{C}\text{---}$ and $\text{---}\text{C}\text{---}\bullet$. This allows us to transform any $m \rightarrow n$ circuits into one of type $0 \rightarrow m + n$ and vice-versa; moreover, any equality that we can apply to the former also applies to the latter. In categorical terms, we say that our semantics, the SMC FinProjStoch , is *compact-closed*.

Proposition 4.5 (Compactness). *There is a one-to-one mapping $(\cdot)^b$ between diagrams of type $m \rightarrow n$ and those of type $0 \rightarrow m + n$; in addition, for any $c; d: m \rightarrow n$, $c = d$ iff $c^b = d^b$.*

The main idea of the completeness proof is to remove all explicit conditioning from $0 \rightarrow n$ circuits to reduce the completeness of the whole syntax to the causal fragment. The main difference with the causal case is that conditioning introduces the possibility of failure: circuits that contain the unsatisfiable constraint $\begin{smallmatrix} \textcircled{1} \\ \textcircled{0} \end{smallmatrix} \text{---}$ and whose semantics is the zero subdistribution. We deal with this special case separately, since any two circuits containing failure are equal:

Lemma 4.6 (Failure). *For any two circuits $c, d : m \rightarrow n$, we have*

$$\begin{array}{c} \perp \\ \hline m \quad c \quad n \end{array} = \begin{array}{c} \perp \\ \hline m \quad d \quad n \end{array}$$

Note that this fact is responsible for the unusual behaviour of our **if-then-else** construct: failure in one branch, even if unevaluated, will imply the failure of the whole program. This is an unavoidable consequence of choosing to identify programs that denote the same distribution up to a renormalisation factor.

Theorem 4.7 (Completeness). *For any two circuits $c, d : m \rightarrow n$ with conditioning, $\llbracket c \rrbracket_\infty = \llbracket d \rrbracket_\infty$ iff $c = d$.*

Proof. By compactness (Proposition 4.5), it is enough to prove the statement for $c, d : 0 \rightarrow n$. We want to show that we can always eliminate all conditioning nodes $\curvearrowright$ to rewrite any diagram $c : 0 \rightarrow n$ into normal form. For this, it is enough to show that any diagram $c : 0 \rightarrow n$ composed (on the right, necessarily) with $\curvearrowright$ is equal to some causal, *i.e.* conditioning-free, circuit.

First, we can assume wlog that $\curvearrowright$ is plugged on the first two wires of c ; if it was not, we could simply reorder them and rewrite the resulting diagram into normal form first. Moreover, by Theorem 3.11, there exists circuits c_0, c_1 and c', c'' in normal form such that:

$$\begin{aligned} \begin{array}{c} \curvearrowright \\ \hline c \quad n-2 \end{array} &= \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c' \quad n-2 \end{array} = \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \\ &= \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \stackrel{\mathbf{A1}}{=} \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \\ &\stackrel{\mathbf{F5;F4}}{=} \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \stackrel{\mathbf{A3}}{=} \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \\ &\stackrel{\mathbf{F4;F5;F}}{=} \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \stackrel{\mathbf{A3}}{=} \begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} \end{aligned}$$

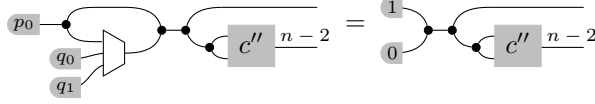
Since c_0, c_1 are themselves in normal form, there exist weights p_0, q_0, q_1 such that

$$\begin{array}{c} \curvearrowright \\ \hline c_0 \quad c_1 \quad c'' \quad n-2 \end{array} = \begin{array}{c} p_0 \\ q_0 \\ q_1 \end{array} \begin{array}{c} \curvearrowright \\ \hline \end{array} \begin{array}{c} \curvearrowright \\ \hline c'' \quad n-2 \end{array}$$

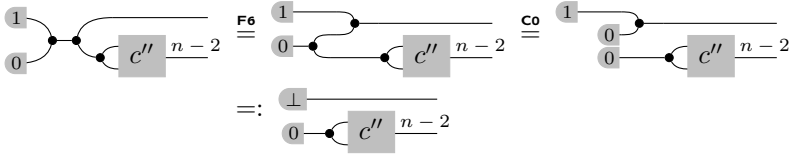
There are two cases to consider. First, if $p_0 q_0 + (1 - p_0)(1 - q_1) \neq 0$, we have

$$\begin{array}{c} p_0 \\ q_0 \\ q_1 \end{array} \begin{array}{c} \curvearrowright \\ \hline \end{array} \begin{array}{c} \curvearrowright \\ \hline c'' \quad n-2 \end{array} \stackrel{\mathbf{F7}}{=} \begin{array}{c} r \\ \hline \end{array} \begin{array}{c} \curvearrowright \\ \hline c'' \quad n-2 \end{array}$$

where $r := \frac{p_0 q_0}{p_0 q_0 + (1-p_0)(1-q_1)}$. Notice that the sub-circuit in the dashed box above is in normal form, so the final circuit is also in normal form. Second, if $p_0 q_0 + (1-p_0)(1-q_1) = 0$, then we either have $p_0 = 1$ and $q_0 = 0$, or $p_0 = 0$ and $q_1 = 1$. Intuitively, in these two cases, the conditioning constraint imposed by $\curvearrowright$ is unsatisfiable, and we get:



Note that this is true in both cases, by commutativity of $\curvearrowright$. Then,



Since any two circuits containing $\perp$ are equal by Lemma 4.6, the last circuit is equal to one in normal form. Thus, any $c : 0 \rightarrow n$ is equal to a circuit in normal form and the same reasoning as in the proof of Theorem 3.11 shows that any two semantically equivalent circuits $0 \rightarrow n$ are equal to the same circuit in normal form, which concludes the proof.

5 Discussion

In summary, we presented a complete axiomatisation of discrete probabilistic program equivalence. We were able to achieve this result by translating a conventional PL into a calculus of string diagrams, representing Boolean circuits extended with primitives denoting Bernoulli distributions and explicit conditioning, which we equipped with a compositional semantics in terms of subdistributions (up to some re-normalisation factor). Then, we gave a complete equational theory for circuits without conditioning, that is, a presentation of the category of stochastic maps between sets that are powers of the Booleans. Finally, we showed how to extend the preceding axiomatisation to all circuits—and therefore all programs—including those that contain explicit conditioning.

Related work. Our choice of syntax and semantics for conditioning is based on the work of Staton and Stein [35,36] where they define the CD-calculus, a PPL with an exact conditioning operation like $(x := y)$ for both Gaussian and discrete probabilistic programming. In the discrete case, they consider two semantics: the one we use here and another in terms of plain subdistributions (without re-normalisation). While they do study some of the equational properties of the CD-calculus in general and of exact conditioning in particular, they do not give a complete axiomatisation for the discrete case. It should be noted that a complete axiomatisation of the Gaussian case appeared recently [37].

Moreover, a presentation of the SMC of stochastic matrices *with the disjoint sum* as monoidal product was given by Fritz [12], with an axiomatisation similar to that of Stone [38]. This already provides a language expressive enough to encode arbitrary distributions over finite sets⁵. However, unlike our circuits, stochastic matrices with disjoint sum cannot meaningfully express correlation between different variables, making it insufficiently expressive as a language for probabilistic programming. Our work is the first to present (a subcategory of) the same underlying category, but *with the cartesian product* as monoidal product.

Finally, this work fits within the broader programme of axiomatising probability theory in Markov categories [13]. Many papers in the same area use string diagrams to generalise and prove key results in statistics and probability theory [17,15,30,14,27,16], without necessarily having an axiomatisation of their domain of interest. Axioms for conditioning in the setting of *partial* Markov categories have also been used to provide a synthetic account of Bayes theorem [10].

Future work. Our paper opens several new directions. Firstly, we believe that axiomatisations of alternative semantics for conditioning can be achieved with a few simple modifications. In particular, the semantics of conditioning in terms of plain subdistributions (without quotient) seems within reach and would be of independent interest. Secondly, we would like to axiomatise KL-divergence between probabilistic circuits/programs using a *quantitative* equational theory [1,30]. This would require us to replace plain equalities with equalities of the form $c \equiv_\varepsilon d$, to represent the fact that the KL-divergence of $\llbracket c \rrbracket$ relative to $\llbracket d \rrbracket$ is less than ε . Interesting new identities can be expressed in this framework, such as

$$\text{p} \text{---} \bullet \text{---} \text{---} \equiv_\varepsilon \begin{array}{c} \text{p} \text{---} \\ \text{p} \text{---} \end{array} \quad \text{where } \varepsilon \geq p \ln \left(\frac{p}{p^2} \right) + (1-p) \ln \left(\frac{1-p}{(1-p)^2} \right).$$

Such a quantitative identity measures the information loss of assuming that some phenomenon can be modelled by a single coin flip when the true distribution consists of two independent coin tosses. The same approach could be used to axiomatise other quantitative measures of program divergence, such as the total variation distance [30].

Acknowledgments.

The authors would like to thank Wojciech Różowski and Dario Stein for many helpful discussions. RP, MTR and AS acknowledge support from ERC grant Autoprobe no. 101002697, ARIA Safeguarded AI program, and EPSRC Standard Grant CLever EP/S028641/1. FZ acknowledges support from EPSRC grant EP/V002376/1, MIUR PRIN P2022HXNSC, and ARIA Safeguarded AI program.

⁵ All of them, since every finite set is in bijection with a disjoint sum of a single generating singleton set.

References

1. Bacci, G., Mardare, R., Panangaden, P., Plotkin, G.: Quantitative equational reasoning. *Foundations of Probabilistic Programming* p. 333 (2020)
2. Baez, J.C., Coya, B., Rebro, F.: Props in network theory. *Theory & Applications of Categories* **33** (2018)
3. Birkhoff, G.: Three observations on linear algebra. *Univ. Nac. Tucumán. Revista A*. **5**, 147–151 (1946)
4. Boisseau, G., Sobociński, P.: String diagrammatic electrical circuit theory. *Electronic Proceedings in Theoretical Computer Science* **372**, 178–191 (Nov 2022). <https://doi.org/10.4204/eptcs.372.13>
5. Boole, G.: *The Laws of Thought* (1854). The Open court publishing company, London, (1854)
6. Cho, K., Jacobs, B.: Disintegration and Bayesian inversion via string diagrams. *Mathematical Structures in Computer Science* **29**(7), 938–971 (2019)
7. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. p. 92–102. ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2491411.2491423>
8. Coecke, B., Kissinger, A.: *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press (2017)
9. Dash, S., Kaddar, Y., Paquet, H., Staton, S.: Affine monads and lazy structures for Bayesian programming. *Proc. ACM Program. Lang.* **7**(POPL) (jan 2023). <https://doi.org/10.1145/3571239>
10. Di Lavore, E., Román, M.: Evidential decision theory via partial markov categories. In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–14. IEEE (2023)
11. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* **15**(3), 358–401 (Apr 2014). <https://doi.org/10.1017/s1471068414000076>
12. Fritz, T.: A presentation of the category of stochastic matrices. *arXiv preprint arXiv:0902.2554* (2009)
13. Fritz, T.: A synthetic approach to markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics* **370**, 107239 (aug 2020). <https://doi.org/10.1016/j.aim.2020.107239>
14. Fritz, T., Gonda, T., Houghton-Larsen, N.G., Lorenzin, A., Perrone, P., Stein, D.: Dilations and information flow axioms in categorical probability. *Mathematical Structures in Computer Science* **33**(10), 913–957 (2023)
15. Fritz, T., Gonda, T., Perrone, P.: De finetti’s theorem in categorical probability. *Journal of Stochastic Analysis* **2**(4), 6 (2021)
16. Fritz, T., Gonda, T., Perrone, P., Rischel, E.F.: Representable markov categories and comparison of statistical experiments in categorical probability. *Theoretical Computer Science* **961**, 113896 (2023)
17. Fritz, T., Rischel, E.F.: Infinite products and zero-one laws in categorical probability. *Compositionality* **2** (2020)
18. Gehr, T., Misailovic, S., Vechev, M.: Psi: Exact symbolic inference for probabilistic programs. In: *Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification*. pp. 62–83. Springer International Publishing (2016)

19. Giry, M.: A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis* pp. 68–85 (1982)
20. Gu, T., Piedeleu, R., Zanasi, F.: A complete diagrammatic calculus for boolean satisfiability. *Electronic Notes in Theoretical Informatics and Computer Science Volume 1-Proceedings of...* (2023). <https://doi.org/10.46298/entics.10481>
21. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. *Commun. ACM* **30**(8), 672–686 (aug 1987). <https://doi.org/10.1145/27651.27653>
22. Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* **4**(OOPSLA), 1–31 (2020)
23. Kozen, D.: Semantics of probabilistic programs. *Journal of Computer and System Sciences* **22**(3), 328–350 (1981). [https://doi.org/https://doi.org/10.1016/0022-0000\(81\)90036-2](https://doi.org/https://doi.org/10.1016/0022-0000(81)90036-2)
24. Kozen, D.: Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.* **19**(3), 427–443 (may 1997). <https://doi.org/10.1145/256167.256195>
25. MacLane, S.: *Categories for the Working Mathematician*. Springer New York (1971). <https://doi.org/10.1007/978-1-4612-9839-7>
26. Moss, S., Perrone, P.: Probability monads with submonads of deterministic states. In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science. LICS '22*, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3531130.3533355>
27. Moss, S., Perrone, P.: A category-theoretic proof of the ergodic decomposition theorem. *Ergodic Theory and Dynamical Systems* **43**(12), 4166–4192 (2023)
28. Narayanan, P., Carrette, J., Romano, W., Shan, C.c., Zinkov, R.: Probabilistic inference by program transformation in Hakaru (system description). In: Kiselyov, O., King, A. (eds.) *Functional and Logic Programming*. pp. 62–79. Springer International Publishing, Cham (2016)
29. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A.: Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **40**(1), 1–50 (2018)
30. Perrone, P.: Markov categories and entropy. *IEEE Transactions on Information Theory* (2023)
31. Piedeleu, R., Zanasi, F.: A string diagrammatic axiomatisation of finite-state automata (2020), <https://arxiv.org/abs/2009.14576>
32. Piedeleu, R., Zanasi, F.: An introduction to string diagrams for computer scientists (2023), <https://arxiv.org/abs/2305.08768>
33. Różowski, W., Kappé, T., Kozen, D., Schmid, T., Silva, A.: Probabilistic guarded kat modulo bisimilarity: Completeness and complexity. In: *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023)
34. Selinger, P.: *A Survey of Graphical Languages for Monoidal Categories*, p. 289–355. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-12821-9_4
35. Stein, D., Staton, S.: Compositional semantics for probabilistic programs with exact conditioning. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–13. IEEE (2021)
36. Stein, D., Staton, S.: Probabilistic programming with exact conditions. *Journal of the ACM* (2023)
37. Stein, D., Zanasi, F., Samuelson, R., Piedeleu, R.: Graphical quadratic algebra. *arXiv preprint arXiv:2403.02284* (2024)

38. Stone, M.H.: Postulates for the barycentric calculus. *Annali di Matematica Pura ed Applicata* **29**, 25–30 (1949)
39. Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican (2016), <https://arxiv.org/abs/1608.05263>
40. Zanasi, F.: Interacting Hopf Algebras-the Theory of Linear Systems. Ph.D. thesis, Ecole normale supérieure de Lyon (2015)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Compositional Shape Analysis with Shared Abduction and Biabductive Loop Acceleration

Florian Sextl¹ , Adam Rogalewicz² , Tomáš Vojnar^{2,3} , and Florian Zuleger¹

¹ TU Wien, Faculty of Informatics, Institute of Logic and Computation, Vienna, Austria,
florian.sextl@tuwien.ac.at

² Brno University of Technology, Faculty of Information Technology, Brno, Czechia

³ Masaryk University, Faculty of Informatics, Brno, Czechia

Abstract. Biabduction-based shape analysis is a compositional verification and analysis technique that can prove memory safety in the presence of complex, linked data structures. Despite its usefulness, several open problems persist for this kind of analysis; two of which we address in this paper. On the one hand, the original analysis is path-sensitive but cannot combine safety requirements for related branches. This causes the analysis to require additional soundness checks and decreases the analysis’ precision. We extend the underlying symbolic execution and propose a framework for *shared abduction* where a common precondition is maintained for related computation branches. On the other hand, prior implementations lift loop acceleration methods from forward analysis to biabduction analysis by applying them separately on the pre- and post-condition, which can lead to imprecise or even unsound acceleration results that do not form a loop invariant. In contrast, we propose *biabductive loop acceleration*, which explicitly constructs and checks candidate loop invariants. For this, we also introduce a novel heuristic called *shape extrapolation*. This heuristic takes advantage of locality in the handling of list-like data structures (which are the most common data structures found in low-level code) and jointly accelerates pre- and post-conditions by extrapolating the related shapes. In addition to making the analysis more precise, our techniques also make biabductive analysis more efficient since they are sound in just one analysis phase. In contrast, prior techniques always require two phases (as the first phase can produce contracts that are unsound and must hence be verified). We experimentally confirm that our techniques improve on prior techniques; both in terms of precision and runtime of the analysis.

Keywords: Shape Analysis · Biabduction

1 Introduction

Over the last two decades, *shape analysis* has proven to be one of the most useful techniques for ensuring memory safety in programs. This kind of analysis focuses on verifying memory-safe handling of linked data structures by representing them as abstract memory *shapes*. Thereby, memory safety can often be verified with fully automatic reasoning for a wide range of data structures. Examples of successful shape analyzers

F. Sextl—Main author, other authors in alphabetical order of their last names

include the tools Predator [10], which has won a number of medals at the well-known SV-COMP competition (see [2,3]), and Infer [4], which has been used for several years to check large code bases at Meta (formerly Facebook).

Among the reasoning principles underlying shape analysis, *biabduction* has the unique ability to enable compositional analysis for open programs (i.e., program fragments) by synthesizing invariants and function contracts consisting of separation logic formulas [25]. The ability to synthesize contracts, i.e., pairs of pre- and post-conditions, allows for a highly modular inter-procedural analysis. In addition, the compositional analysis with biabduction enjoys what Calcagno et al. [5,6] have called “graceful imprecision”, i.e., the analysis will find useful results for some parts of a program, even if it introduced imprecisions for other parts. In contrast, closed program analyses are likely to build up such imprecisions and fail, even if they could handle further parts of the program otherwise. Due to this and due to not requiring programmers to develop verification harnesses, biabduction-based shape analysis is often considered to be more useful for large-scale verification compared to techniques for closed programs, which are advantageous for smaller, self-contained programs. However, this advantage comes at the cost of more complex computation principles as well as generally less expressive abstract shapes. Moreover, existing biabduction-based shape analyses can compute unsound results and, thus, require a second analysis phase to filter out these results.

The highly path-sensitive analysis proposed by [5,6] does not work well with branching if the branch to be taken cannot be determined purely from the pre-condition of the analyzed function (e.g., because it depends on user input or because the used logical fragment cannot express the dependency sufficiently), see [6, section 4.3]. We call these cases *non-determinable branching*. The problem with these cases arises since maintaining different pre-conditions for each program path is, in general, insufficient because the only sound precondition might consist of a combination of these. Thus, Calcagno et al. suggested a heuristic procedure for merging pre-conditions, but their approach may fail to compute any valid pre-conditions (see Section 2.1) at all. In contrast, we present a novel technique of *shared abduction*, which allows for sound pre-condition computation across program branches. The technique extends biabductive symbolic execution by tracking which program locations share which pre-condition requirements. Due to this, shared abduction circumvents the need for a verification phase for programs with arbitrary branching, and, at the same time, can infer non-trivial contracts in more cases than the traditional analysis (since it does not give up on the cases where some sharing of information between branches is necessary).

Symbolic analyses generally require loop acceleration techniques to allow the analyses to reach a fixed point. In shape analysis, this acceleration replaces concrete pointers with more abstract, typically inductive shape predicates such as list segments. The analysis by [5] lifts such abstraction to the setting of biabductive analysis by applying abstraction separately to the pre- and the post-condition. However, such a direct lifting is not guaranteed to result in a sound loop invariant (see [6, section 4.3] or [29, Appendix F.1]). In contrast, we introduce a novel *biabductive loop acceleration* scheme that constructs candidate loop invariants after analyzing the loop body once. This step allows us to verify the soundness of the candidate invariant explicitly through another symbolic execution of the loop body. To construct the candidate invariant, we intro-

duce a novel *shape extrapolation* heuristic, which exploits the locality of typical data structure traversals to find fitting abstract shape predicates.

Even though the two-phase approach is easy to implement as the two phases primarily differ in which biabduction rules are used, and even though most related works rely on the second phase (see [13,12]), it is quite natural to wonder whether the overhead of the two phases can be reduced. Overcoming this overhead for the broader family of biabduction-based shape analyses is exactly the goal of this paper. Thereby, our shared abduction technique avoids the unsoundness problem for non-determinable branching and circumvents the need for the second phase (at the same time, producing more non-trivial contracts than the previous approaches). In addition, our loop acceleration approach only requires us to check the extrapolated invariant for soundness, but this check is much more local and less costly than the second analysis phase.⁴ Our approach to constructing the invariant is heuristic and, hence, not always applicable, but the underlying invariant check is still guaranteed to assert soundness. In addition, our experiments demonstrate that our approach can significantly improve the efficiency, i.e., reduce the needed number of loop iterations and lower the runtime compared to the two-phase biabduction architecture in practice. We conjecture that this is because programmers do commonly write loops in a way compatible with our approach.

Main Contributions. The main contributions of this work in the context of analyzing sequential, non-recursive programs are the following:

- A novel sound analysis for loop-free code based on *shared abduction* (Section 4).
- A novel sound way to construct and check loop invariants as part of biabductive shape analysis via *biabductive loop acceleration*. It uses a novel heuristic to exploit locality via *shape extrapolation* on list-manipulating programs (Section 5).
- Formal proofs of shared abduction and biabductive loop acceleration being sound (Theorems 1 and 2 with proofs in [29, Appendix D]).
- An experimental evaluation based on a proof-of-concept implementation applied to a number of small-scale but rather challenging programs, including real-life library functions, that show the superiority of our approach with regard to runtime and completeness compared to established analyzers (Section 6).

General Limitations. Our acceleration approach is currently limited to programs manipulating various kinds of lists (singly or doubly linked, possibly circular, nested, and intrusive). While this restriction coincides with prior work [6,13], we hope that exploiting locality for loop acceleration will apply to further data structures (such as trees), but we must leave this for future work. Furthermore, we only focus on non-recursive programs, following most previous biabductive shape analysis approaches. Qin et al. [22] introduced an extension to handle recursion via a fixed point computation of the function contract, but this is orthogonal to our work. Moreover, our prototype tool is focused on low-level C code, which rarely contains recursion anyway.

An extended version of this paper with the full appendix can be found as [29].

⁴ The traditional approaches to biabduction, such as [13,12], will analyze each loop at least twice (to get to a fixpoint) in each of the phases, i.e., each loop is analyzed *at least four times*, but often even more (see table 2). On the other hand, our approach may soundly find a loop invariant within one analysis phase, which analyses the loop in general only twice.

$\text{true for read} \leq 0 \vee \text{in\_mode} \geq 2 \quad | \quad hd \mapsto \ell_1 * out \mapsto \ell_4 \text{ for in\_mode} = 0$
 $\text{curr} \neq hd : \text{curr} \mapsto \ell_3 * out \mapsto \ell_4 \text{ and } \text{curr} = hd : \text{lst} \mapsto \ell_2 * out \mapsto \ell_4 \text{ for in\_mode} = 1$

Fig. 1: Insufficient candidate pre-conditions for `user_choice`

2 Motivation

2.1 Cross-Branch Abduction Sharing

```

int user_choice(node *hd, node *lst,
               node *curr, node *out) {
  int in_mode = 0;
  int read = scanf("%d", &in_mode);

  if (read <= 0) {
    return -1;
  } else if (in_mode == 0) {
    memcpy(out, hd, ...);
  } else if (in_mode == 1) {
    if (curr != hd) {
      memcpy(out, curr, ...);
    } else {
      memcpy(out, lst, ...);
    }
  }
  return in_mode;
}

```

Listing 1: Non-determinable branching

Listing 1 shows a program fragment that works on a data node based on user input read from the command line via `scanf`. Based on this, `user_choice` takes the user input and calls `memcpy` with the corresponding arguments. The exact invocation depends on the user input,⁵ which is modeled by a non-deterministic choice in the analysis, and, in the case of `in_mode = 1`, also depends on whether `curr = hd`. We call these kinds of branching *non-determinable*, since the branch taken at runtime can't be determined from the function parameters alone.

Problem. Non-determinable branching is difficult to handle for a path-sensitive biabduction-based shape analysis as proposed in [6,5]. This is because such an analysis will generate one precondition per program branch in Figure 1, expressed with standard separation logic connectives.⁶

We note that the preconditions can take into account branching conditions that depend on the function's arguments, e.g., the preconditions in Figure 1 contain the predicates $\text{curr} = \text{hd}$ and $\text{curr} \neq \text{hd}$. However, non-determinable branching, such as for $\text{in\_mode} = 0$, cannot be modeled in terms of the function's arguments, and hence such conditions can never be part of a precondition. Then the problem arises that the different branches require different memory locations to be allocated (note the different pointers arguments to `memcpy`), e.g. $hd \mapsto \ell_1 * out \mapsto \ell_4$ for the branch with $\text{in\_mode} = 0$. However, due to the non-deterministic input, none of the required allocations for one branch guarantee a memory-safe execution for all user inputs.

Previous Solutions. This problem has already been noticed in the original work [6,5] and was partially addressed by a heuristic that would combine pre-conditions such that they could cover move branches. This heuristic has been implemented as an optional strategy in the Abductor tool and was subsequently made the default in the Infer tool.

⁵ We chose user input as an easy to understand example of non-determinable input. Other cases of such input includes IO operations such as incoming network traffic or reading from a file.

⁶ We use “:” to separate the formulas' pure and spatial parts (if any). In contrast to the program variables *hd*, etc., the ℓ_i variables are purely logical and implicitly universally quantified. We write program variables in formulas in *italic* and otherwise in typewriter font.

However, this heuristic does not guarantee that the found pre-conditions are sound, and it can easily miss safe pre-conditions even for simple loop-free code. Indeed, for the example above, the heuristic finds the combined pre-condition $curr \neq hd : hd \mapsto \ell_1 * lst \mapsto \ell_2 * curr \mapsto \ell_3 * out \mapsto \ell_4$, but it does not produce a sound pre-condition for the case $curr = hd$. We also remark that the heuristic is quite fragile as renaming the variable hd to $first$ enables the heuristic to find a safe pre-condition for the case $curr = hd$ in Infer while leading to a crash for Abductor.⁷

We finally note that recent work [33], developed concurrently with our approach, also addresses the problem of unsound pre-conditions for branching programs. They introduce a specialized operator called *tri-abduction*, which generalizes the setting of *bi-abduction* to simultaneously compute a combined pre-condition for two branches. While this operator offers more precision than the (classical) biabduction operator we rely on in this paper, it is unclear how to build a realistic symbolic execution based on the tri-abduction operator. To this date, such an analysis has only been sketched but not implemented. We comment more on the relationship to our approach in Section 7.

Shared Abduction. The fundamental problem discussed above is that the different program paths cannot be analyzed in isolation; instead, we must combine their preconditions. That is, biabduction-based analyzers need to track which program configurations can be reached from the same initial configuration and synchronize the abducted requirements. Moreover, the analysis needs to be precise in tracking which configurations are allowed to exchange such information – otherwise, inconsistencies can be introduced by exchanging information among independent program points. Our solution is, therefore, to track exactly which program configurations can be reached from a common pre-condition and explicitly share newly found requirements with all of these configurations (and only such configurations). Section 4 introduces how this technique, which we call *shared abduction*, allows sound handling of all kinds of branching.

Our technique has the advantage of being lightweight and easily implementable on top of an existing biabductive analysis. In the example above, our analysis first abduces the precondition $hd \mapsto \ell_1 * out \mapsto \ell_4$ for the case $in\_mode = 0$. The analysis then proceeds with the branch for $in\_mode = 1$, making a case distinction on $curr = hd$. Shared abduction retains the required allocation $hd \mapsto \ell_1 * out \mapsto \ell_4$ for both cases as this requirement is already part of the shared precondition. Then, by analyzing the nested branches, the requirements $curr \neq hd : hd \mapsto \ell_1 * curr \mapsto \ell_3 * out \mapsto \ell_4$ and $curr = hd : hd \mapsto \ell_1 * lst \mapsto \ell_2 * out \mapsto \ell_4$ are computed. We note that this guarantees the soundness of the found pre-condition and its completeness regarding the branching, thus outperforming the previous heuristic.

2.2 Shape Extrapolation for Biabductive Acceleration

Our second contribution aims at the analysis of loops, which generally requires accelerating the symbolic execution to allow the analysis to reach a fixed point.

⁷ We observed this behavior with the commit f93cb281edb33510d0a300f1e4c334c6f14d6d26 found at <https://github.com/facebook/infer> and the publicly available Abductor release at <http://www0.cs.ucl.ac.uk/staff/p.ohearn/abductor.html>.

Problems. The prior technique for loop acceleration, proposed by [5,6] and adopted in [13], separately abstracts the pre- and post-condition with no other information than the formulas themselves taken into account. Intuitively, after analyzing some loop iterations and applying the abstraction operator, the obtained formulas will stabilize, and a fixed point is reached. Thereby, the abstraction follows the intuitive principle of collecting linked memory blocks with a similar layout into a single abstract shape predicate. In the context of simple singly-linked lists, this means that the abstraction procedure scans the formula for points-to predicates $x \mapsto \ell$ and $\ell \mapsto z$, linked by a location ℓ (i.e., the target of the first predicate contains the address of the second), or a linked list segment $ls(x, \ell)$ and a points-to $\ell \mapsto z$, respectively. Abstraction then replaces these predicates with the single predicate $ls(x, z)$. However, abstraction cannot be applied when there is a program variable y that references ℓ , e.g., as $y = \ell$. This is not supported since it would lose the information that variable y is allocated (note that ℓ does not occur in $ls(x, z)$ anymore). More generally, abstraction cannot be applied if there is a program variable y whose value depends on ℓ , such as $y = v \wedge \ell \mapsto v$. While the abstraction principle is intuitive, there are also multiple drawbacks, which we discuss next.

```
void weighted_sum(o_node *o,
                  long *sum) {
  while (o != NULL) {
    i_node *i = o->inner;
    while (i != NULL) {
      *sum = (*sum) +
        (o->wgt * i->elem);
      i = i->next;
    }
    o = o->next_o;
  }
}
```

Listing 2: Nested list traversal

```
void traverse_skip_two(node *list) {
  node *tmp = list->next->next;
  while (tmp != NULL) {
    tmp = tmp->next;
  }
}
```

Listing 3: Offset list traversal

(1) For the example in Listing 2, Abductor, Infer, and Broom do not reach a fixed point for the inner loop and thus cannot synthesize any contract. This is because the value pointed to by *sum* after n loop iterations is $\ell_{sum} + \ell_1 \cdot \ell_w + \dots + \ell_n \cdot \ell_w$, where ℓ_w is the value pointed to by *o.wgt* and the ℓ_i are the *elem* values of the list nodes traversed so far. Thus, the dependence on the values ℓ_i blocks abstraction (as described above). We note that the design of a stronger abstraction operator is not straight-forward because we also need to track values in memory precisely as they could be relevant for memory accesses based on pointer arithmetic in other parts of the program.

(2) A formula-based abstraction operator can easily lose too much information. For example, Abductor, Infer, and Broom fail for the simple example in Listing 3. The reason is as follows: The abstraction operation (as described above) contracts pointer chains of length at least two into a list segment, resulting in the formula $list \neq NULL : ls(list, NULL)$. This predicate describes a non-empty list segment with at least one node. However, this formula does not suffice as a pre-condition that guarantees memory safety because *traverse_skip_two* requires a list of length ≥ 2 as input.

(3) Furthermore, acceleration based on abstraction (as described above and implemented in Abductor, Infer, and Broom) can be highly inefficient. In general, every loop will require at least two (often three) analysis iterations, as abstraction can often only be applied after the second loop iteration and a fixed point can only be checked for after another iteration. In the presence of inner loops, such as for the example in Listing 2, this quickly multiplies, e.g., amounting to nine analysis iterations for the inner loop in Listing 2 just for the first analysis phase.

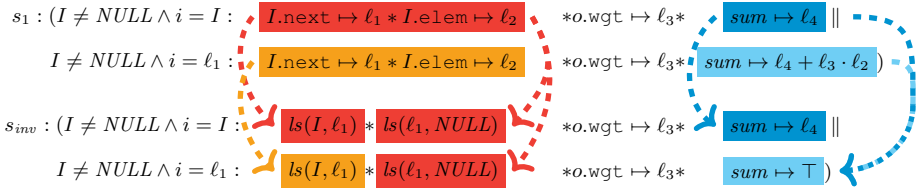


Fig. 2: State s_1 after the first loop iteration analysis and the constructed candidate invariant s_{inv} , with color-coded arrows showing the information flow between different subformulas

The problems described above are mostly related to the direct application of abstraction for acceleration and its missing ability to take into account more information about the loop, e.g. the observation that inductive data structures are often traversed one step at a time. For example, in Listing 2, the nested linked list is traversed in such a way to compute the weighted sum of the elements in the inner lists. It is apparent that each of the two loops operates on a local, shifting view of the respective traversed list plus some context. For the inner loop, this means that the loop only operates on a unique `i_node` at a time while also accessing the same variables `sum` and `o->wgt` in each iteration. Similarly, the outer loop only operates on one `o_node` at a time. These shifting views on the traversed shapes are akin to “local actions” (see [7]).

Biabductive Loop Acceleration. This observation allows us to extrapolate what the analysis abduces from a single iteration to arbitrarily many iterations and directly compute a candidate loop invariant if applicable. We call this heuristic *shape extrapolation*. It is part of our *biabductive loop acceleration*, which consists of three main steps: First, we use the analysis result of a single iteration to obtain locality information about the shape and the context; second, we use this information to extrapolate the shape to an abstract one; third, we check that the heuristically constructed state is a sound invariant.

In the case of `weighted_sum`, after the first iteration of the inner loop, the analysis finds the state s_1 depicted in Figure 2, consisting of a pre- and post-condition separated by $\parallel$. Our analysis then partitions the pre- as well as the post-condition into a shape and a context part, where the shape part is $I.next \mapsto \ell_1 * I.elem \mapsto \ell_2$, and the context is $o.wgt \mapsto \ell_3 * sum \mapsto \ell_4$ for the pre-condition and $o.wgt \mapsto \ell_3 * sum \mapsto \ell_4 + \ell_3 \cdot \ell_2$ for the post-condition. The following heuristic obtains this partitioning: We consider the changed variables (here `i`, `sum`) whose value moved to some pointer value (here `i` whose value moved to `i->next`). The predicates associated with these variables are then put into the shape parts and the others into the context.

Based on this partitioning, our procedure directly constructs a (candidate) loop invariant s_{inv} (see Figure 2). Thereby, our procedure accelerates the shape part of the pre- as well as the post-condition (here we obtain in both cases the list segment $ls(I, \ell_1)$, with internal next pointer `next` and data field `elem`). Intuitively, these predicates correspond to the loop iterations up to the current point. In addition, we add predicates $ls(\ell_1, NULL)$ to the shape part of s_{inv} , for both the pre- and post-condition, which are taken as the accelerated predicate $ls(I, \ell_1)$ of the pre-condition of s_1 , where I has been replaced with ℓ_1 , which is the current value of `i`, and ℓ_1 has been replaced with `NULL`, which has been obtained from the loop condition. Intuitively, these predicates corre-

spond to the future loop iterations up to the loop's termination. We refer to the red and orange colors in Fig. 2 to illustrate the information flow. The context part of s_{inv} keeps the context of s_1 , except that our procedure abstracts the value of sum in the post-condition (with the unknown value $\top$) because it cannot be tracked precisely. Next, our procedure checks that s_{inv} is indeed a loop invariant, which requires one symbolic execution of the loop body and an entailment check.

Finally, based on the loop invariant our analysis constructs a contract that abstracts the inner loop and that can be used for the analysis of the outer loop:

$$\begin{aligned} (i = I : ls(I, NULL) * o.wgt \mapsto \ell_3 * sum \mapsto \ell_4 \parallel \\ i = NULL : ls(I, NULL) * o.wgt \mapsto \ell_3 * sum \mapsto \top). \end{aligned}$$

Based on this contract, our procedure then also accelerates the outer loop in a similar fashion. Lastly, we note that our procedure requires exactly two iterations per loop (in sum four): one to analyze the effects of the loop and a second to check whether the constructed state is a loop invariant (as opposed to the nine iterations in sum mentioned earlier for the traditional acceleration). Similarly, our approach fails fast if the constructed candidate invariant is unsound instead of requiring a second analysis phase with many more analysis steps.

3 Preliminaries

We present our new techniques for a simple but standard setting that is described in the following since it does not require any specific logic fragment or biabduction method.

3.1 Programming Language and Memory Model

Let Var be a countably infinite set of variables and Val be a countably infinite set of values containing the value $\top$. Furthermore, let Fld be a finite set of field names and $Loc \subseteq Val$ be the set of memory locations such that $NULL \in Loc$.

Definition 1 (Programming Language). *We base our description on a C-like programming language. The language comprises standard expressions, statements for reading ($x_1 = *x_2.f$) and writing ($*x_1.f = x_2$) through pointers (with the C-like syntactic sugar of $x \rightarrow f$ for $*x.f$), an additional non-determinism operator $?$, as well as implicit control flow statements ASSUME and ASSERT. Moreover, we include the C-like commands for memory (de-)allocation ALLOC and FREE.*

Functions have a function name f , a list of argument variables $a_1, \dots, a_n$, $n \geq 0$, and a function body $body_f$ that consists of a *control flow graph*. We represent *control flow graphs* (CFGs) as a tuple $(V, E, entry_f, exit_f)$ such that V is a set of program locations with $entry_f, exit_f \in V$, and $E \subseteq V \times stmt \times V$ is a set of edges between program locations labeled with program statements. A *trace* t of a CFG is an alternating sequence $[v_0, st_1, v_1, \dots, st_n, v_n]$, $n \geq 0$, of vertices $v_i \in V$ and statements $st_{i+1} \in stmt$ following edges $(v_i, st_{i+1}, v_{i+1}) \in E$.

Note that CFGs as defined here can be used to model arbitrary branching and looping constructs (such as `if-then-else` and `while`). We will further assume:

1. Each function f is either loop-free or consists of a single loop such that the loop header is $entry_f$; i.e., we require that the CFG of f is either acyclic or all back-edges of f (the edges returning to a loop header) return to $entry_f$. This assumption is w.l.o.g. as loops that are embedded in a bigger context can be represented by calls to a function whose body is precisely the loop.⁸
2. The programs do not contain (mutually) recursive functions.
3. Each vertex $v \in V$ has at most two outgoing edges in E .

Definition 2 (Program Configuration). A program configuration $cnf \in Config$ is either a pair (S, H) consisting of a stack S and a heap H or the dedicated *err* configuration. The stack $S : Var \rightarrow Val$ is a partial map from variables to values. The heap $H : (Loc \times Fld) \rightarrow_{fin} Val$ partially maps finitely many pairs of memory locations and field names into values.

The semantics of the programming language is standard (see [29, Appendix A] for more details). We use the notation $(cnf_1, st) \rightsquigarrow cnf_2$ to denote that a program reaches a configuration cnf_2 from a configuration cnf_1 by executing a statement st . The semantics of traces is defined as the transitive closure $\rightsquigarrow^*$ with regard to the statements in the trace $(cnf_i \in Config)$:

$$\begin{aligned}
 (cnf_1, [v_0]) &\rightsquigarrow^* cnf_1, \\
 (cnf_1, [t, v_1, st_2, v_2]) &\rightsquigarrow^* cnf_3 \text{ if } (v_1, st_2, v_2) \in E \wedge (cnf_1, [t, v_1]) \rightsquigarrow^* cnf_2 \\
 &\quad \wedge (cnf_2, st_2) \rightsquigarrow cnf_3.
 \end{aligned}$$

3.2 Separation Logic

Next, we introduce a simple separation logic (*SL*) fragment that is suited for biabduction-based shape analysis. Even though most shape analyses in recent literature are based on more sophisticated fragments, this simple fragment suffices to define our central contributions, which can be easily lifted to more powerful separation logic fragments as well (indeed, we use a more expressive fragment in our later presented experiments). The formulas of *SL* are based on symbolic heaps [1].

We differentiate program variables in *PVar* that occur in the program (and are uniquely identifiable for each function) and logical variables in *LVar* that only occur in *SL* formulas. We enforce a *normal form* [19, 13] where program variables only occur once per formula (as part of Π_P in Figure 3), such that they are uniquely identified by a single expression at any point. Further, the dedicated logical variable $return_f$ denotes the return value of a function f , if any.

Definition 3 (Separation Logic). The separation logic fragment *SL* contains the standard connectives of separation logic and an inductive predicate ls denoting a singly-linked list segment. Figure 3 shows the full syntax of *SL*. Symbolic heaps φ distinguish between spatial (Σ) and pure parts (Π), as well as program variable equalities Π_P , and are combined disjunctively (Δ).

⁸ More complex cyclic control flow, e.g. describing the common `break/continue/goto` statements can also be emulated by introducing auxiliary out parameters which are then used to guide the control flow.

Based on this, we define *contracts* for functions in the programming language to be pairs of formulas from SL where we call the parts of the pair a *pre-condition* and a *post-condition*, respectively, and denote them as $c.pre$ and $c.post$ for a contract c . Basic contracts for all statements in the language defined in SL can be found in [29, Appendix B]. Formulas from SL are evaluated against program configurations with the judgement $\models$. We note that we chose the standard semantics for $*$ and $\mapsto$. Furthermore, we also use SL 's *entailment* judgement, written $P \vdash Q$.

We denote with $P[x/y]$ the formula P with variable $y \in LVar$ substituted with x or, if $y \in PVar$, the formula updated with $y = x$ as part of Π_P , respectively. We often denote a formula $\Pi_P \wedge \Pi : \Sigma$ by only $\Pi_P \wedge \Pi$ or $\Pi_P : \Sigma$ if $\Sigma = \mathbf{emp}$ or $\Pi = \mathbf{true}$, respectively. Also, we denote the composition of formulas $\varphi_1 = \Pi_P \wedge \Pi_1 : \Sigma_1$ and $\varphi_2 = \Pi_P \wedge \Pi_2 : \Sigma_2$ as $\varphi_1 * \varphi_2 := \Pi_P \wedge \Pi_1 \wedge \Pi_2 : \Sigma_1 * \Sigma_2$.

Definition 4 (Abstraction). An abstraction function $\alpha : SL \rightarrow SL$ takes a formula in SL and returns a potentially different formula such that it abstracts a given formula P such that $P \vdash \alpha(P)$.

Example 1. An abstraction procedure α as described by [8] abstracts consecutive pointer chains into list segments, e.g., for $PVar = \{x\}$:

$$\alpha(x = a \wedge a.next \mapsto b * b.next \mapsto c) \equiv x = a \wedge ls(a, c).^9$$

3.3 Biabduction-Based Shape Analysis

Definition 5 (Biabduction). Biabduction is the process of solving a query $P * \boxed{M} \vdash Q * \boxed{F}$ for given SL formulas P and Q by computing an *antiframe* (or *missing part*) M and a *frame* F such that the entailment is valid.

We are only interested in solutions for M that do not contradict P , as otherwise, the entailment would be trivially valid. A *biabduction procedure* is then an algorithm that, given two formulas, either computes a fitting frame and anti-frame or fails. The steps to compute a frame and anti-frame are called *frame inference* and *abduction*, respectively.

For the sake of saving space, we do not develop a full biabduction procedure here but refer the reader to [6,5,13] for detailed descriptions.

Definition 6 (Analysis States). An analysis state s is an intermediate contract $(P \parallel Q)$ where $P \in \varphi$ and $Q \in \Delta$. To distinguish these from finished contracts, we call P the *candidate pre-condition* ($s.pre$) and Q the *current post-condition* ($s.curr$). In analysis

$$\begin{aligned} \varepsilon &:= NULL \mid k \in \mathbb{N} \mid \ell \in LVar \mid \mathbf{unop} \ \varepsilon \\ &\quad \mid \varepsilon_1 \ \mathbf{binop} \ \varepsilon_2 \mid \top \\ \oplus &:= = \mid \neq \mid \leq \mid \geq \mid < \mid > \\ \Sigma &:= x.f \mapsto \varepsilon \mid \Sigma_1 * \Sigma_2 \mid ls(x, \varepsilon) \mid \mathbf{emp} \\ \Pi &:= \Pi_1 \wedge \Pi_2 \mid \mathbf{true} \mid \varepsilon_1 \oplus \varepsilon_2 \\ \Pi_P &:= \bigwedge_{x \in PVar} x = \varepsilon \quad \varphi := \Pi_P \wedge \Pi : \Sigma \\ \Delta &:= \varphi \vee \Delta \mid \varphi \end{aligned}$$

Fig. 3: The syntax of SL

⁹ As this step loses information about b , it is only applied in contexts in which b is not relevant otherwise. See, e.g., [16]. This is the case here since $b \notin PVar$ and $\nexists y \in PVar. \Pi_P \vdash y = b$.

states, each function argument $a_i \in PVar$ is associated with an anchor variable $A_i \in AnchVar \subseteq LVar$ (in upper case) denoting its value at entry _{f} . We omit equalities of the form $x = X$ from Π_P if they are not relevant. These anchor equalities also do not appear in finished contracts.

Biabductive Symbolic Execution Step. Let there be an analysis state $(P \parallel Q)$ at a program location l for a statement st with contract $(L \parallel R)$ and a location l' such that $(l, st, l') \in E$. Then st can be symbolically executed by solving the biabduction query $Q * \boxed{M} \vdash L * \boxed{F}$ resulting in the new analysis state $(P * M \parallel R * F)$. As in [6,5,13], we require that (1) $var(M) \subseteq LVar$ and that (2) $P * M$ is satisfiable. If such an M does not exist, we say that the biabduction *fails*.

Definition 7 (Biabduction-based Shape Analysis). A basic biabduction-based shape analysis $A_{B,\alpha}$ uses a biabduction procedure B and an abstraction procedure α to analyze programs in our programming language. Thereby, it analyzes the functions bottom-up along the call tree, starting from its leaves. In each step, the analysis takes an analysis state and symbolically executes the next statement from it by updating the state accordingly. In the case of multiple contracts, the analysis has to determine the applicable ones and continue from each of these.

Furthermore, the analysis runs for a function f until it reaches a fixed point, i.e., until no new analysis states are computed. A common way to check for this condition is to check whether new analysis states entail already computed ones. To enforce termination, $A_{B,\alpha}$ also applies α to abstract the analysis states at loop heads. Finally, the pairs of candidate pre-conditions and current post-conditions forming the analysis states that reached $exit_f$ become its contracts.

We now fix an arbitrary, but correct biabduction-based shape analysis $A_{B,\alpha}$, which we extend in the following sections.

Definition 8 (Soundness of Analysis States). An analysis state $s = (P \parallel Q)$ is called *sound* for a trace t , written as the Hoare triple $\{P\} t \{Q\}$, iff

$$\forall cnf, cnf' \in Config. \quad cnf \models P \wedge (cnf, t) \rightsquigarrow^* cnf' \implies cnf' \neq err \wedge cnf' \models Q.$$

Similarly, a function contract $c = (P, Q)$ is *sound* for $body_f$, written $\{P\} body_f \{Q\}$, iff $\{P\} t \{Q\}$ holds for all traces $t = [entry_f, \dots, exit_f]$ through $body_f$.

The initial analysis state s_0 for function f has $s_0.pre = true$, $s_0.curr = \bigwedge \{x = X \mid x \in PVar \wedge X \in AnchVar\}$, which denotes that each program variable has a fixed but initially unrestricted value (anchor) at the start of f .

Handling of ASSUME. Following the seminal work [6,5] and the more recent [13], we define biabductive shape analysis to split its states at branching points according to the branching condition. As the literature contains sufficient explanations of this mechanism (called assume-as-assume and assume-as-assert), we only give a brief intuition here. If the branching condition can be expressed in terms of the function arguments, i.e., if the branch taken can be statically determined purely from the function arguments, the analysis includes the two cases into the pre-conditions of the resulting states. This

treatment is equivalent to handling the branching condition's ASSUME statements as if they were ASSERT statements instead. Otherwise, the analysis states for the branches have the same pre-condition, and the branching condition cases are only added to the corresponding post-conditions.

4 Sound Branching Analysis with Shared Abduction

```

1 int nested(node *x, node *y,
              node *z) {
2   if (?) {
3     if (y != NULL) {
4       return y->data;
5     } else {
6       return z->data;
7     }
8   } else {
9     return x->data;
10  } }

```

Listing 4: Nested branching

As the example in Listing 1 is rather convoluted, we introduce the simpler Listing 4 to show how exactly the technique works. There, the function `nested` loads from one of the three pointer arguments, depending on a non-deterministic condition ? on Line 2 and a deterministic one on Line 3. Regardless of the values of the function arguments, an execution can either take the then or the else branch of the outer if-then-else.

Therefore, the original analysis simply splits the analysis state without abducting any pre-condition. In contrast, the branches of the inner if-then-else can be distinguished by whether the argument `y` is initially a null pointer, leading the analysis to abduce different pre-conditions for each branch. Altogether, the classical biabduction-based shape analysis will find three unsound contracts for the function, one for each possible code path, similar to the following:

$$\begin{aligned}
 & (x.data \mapsto \ell_1 \parallel \text{return}_{\text{nested}} = \ell_1 : x.data \mapsto \ell_1) \\
 & (y \neq \text{NULL} : y.data \mapsto \ell_2 \parallel y \neq \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_2 : y.data \mapsto \ell_2) \\
 & (y = \text{NULL} : z.data \mapsto \ell_3 \parallel y = \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_3 : z.data \mapsto \ell_3)
 \end{aligned}$$

As introduced in Section 2.1, our new technique overcomes this unsoundness issue and shares requirements abducted with related analysis states. To guarantee that the requirements are only shared with actually related analysis states, we introduce so-called *extended analysis states* or *worlds* for short.

Definition 9 (Worlds). Worlds comprise a shared pre-condition P and multiple current post-conditions $Q_i^{l_i}$ at possibly different program locations l_i : $(P \parallel Q_0^{l_0} \vee \dots \vee Q_n^{l_n})$

We stress the seemingly small but crucial difference between the current post-conditions used in our notion of worlds and the previously defined abstract states: the latter are, in general, also allowed to use disjunctions but are missing the labeling by program locations (allowing the disjuncts to be associated with different program paths). Moreover, worlds do not require the logic itself to contain disjunctions but merely simulates them with its structure.

Definition 10 (Soundness of Worlds). A world $w = (P \parallel Q_0^{l_0} \vee \dots \vee Q_n^{l_n})$ is sound for a trace $t = [v_0, \dots, v_n]$, written $\{P\} t \{Q_0^{l_0} \vee \dots \vee Q_n^{l_n}\}$, iff $\forall \text{conf}, \text{conf}' \in \text{Config}. \text{conf} \models P \wedge (\text{conf}, t) \rightsquigarrow^* \text{conf}' \implies \text{conf}' \neq \text{err} \wedge \exists i. l_i = v_n \wedge \text{conf}' \models Q_i$.

Definition 11 (Shared Abduction). *If the analysis finds a non-empty anti-frame for any of the world's current post-conditions Q_i , it is added to the shared pre-condition P and to all other current post-conditions. We call this shared abduction. This step is motivated by the frame rule of separation logic and works as follows: If M and F are the solution to the biabduction query $Q_i * \boxed{M} \vdash L * \boxed{F}$ where $(L \parallel R)$ is the contract of the statement st that is the label of the edge $(l_i, st, l_{i'})$, then the world $(P \parallel Q_0^{l_0} \vee \dots \vee Q_i^{l_i} \vee \dots \vee Q_n^{l_n})$ gets updated to:*

$$(P * M \parallel (Q_0 * M)^{l_0} \vee \dots \vee (Q_i * M)^{l_i} \vee \dots \vee (Q_n * M)^{l_n} \vee (F * R)^{l_{i'}}).$$

Analysis with Worlds. Whereas analysis states can be split at branching statements by simply duplicating them and adding the respective assumptions, world splits need to be treated differently. The two branches must share their abducted pre-conditions if the branch taken cannot be determined from the initial program state. Therefore, such a branching point with condition c at a location l_i with two successor locations l_j and l_k for a current post-condition Q_i leads to transforming the world from $(P \parallel Q_0^{l_0} \vee \dots \vee Q_i^{l_i} \vee \dots \vee Q_n^{l_n})$ to $(P \parallel Q_0^{l_0} \vee \dots \vee Q_i^{l_i} \vee \dots \vee Q_n^{l_n} \vee (Q_i \wedge c)^{l_j} \vee (Q_i \wedge \neg c)^{l_k})$, where two new post-conditions are added to the world.

In contrast, if the branch can be determined from the initial program state, the whole world must be split into two to ensure shared abduction works correctly. This means that the world at the branching point is exchanged with two new worlds:

$$(P \wedge c \parallel (Q_0 \wedge c)^{l_0} \vee \dots \vee (Q_i \wedge c)^{l_i} \vee \dots \vee (Q_n \wedge c)^{l_n} \vee (Q_i \wedge c)^{l_j}),$$

$$(P \wedge \neg c \parallel (Q_0 \wedge \neg c)^{l_0} \vee \dots \vee (Q_i \wedge \neg c)^{l_i} \vee \dots \vee (Q_n \wedge \neg c)^{l_n} \vee (Q_i \wedge \neg c)^{l_k}).$$

Theorem 1 (Loop-free Soundness with Worlds). *Let $A_{B,\alpha}$ return only sound contracts for functions without branching. Further, let $A'_{B,\alpha}$ be the biabduction-based shape analysis obtained by extending $A_{B,\alpha}$ to use worlds as its analysis states and to apply shared abduction. Then, the contracts computed by $A'_{B,\alpha}$ for loop-free functions are sound. **Proof:** See [29, Appendix D.1].*

Example 2. With these ideas, the function in Listing 4 can be analyzed as follows. At the start of the function, the world is equivalent to an initial analysis state:

$$(\text{true} \parallel (x = X \wedge y = Y \wedge z = Z)_0^1)$$

We denote program locations with their respective lines in the listing and only show the current post-conditions with the highest line number for each branch. Furthermore, we add subscripts to identify the different current post-conditions and worlds uniquely. At the outer `if-then-else`, the current post-condition is split into two as the branching condition cannot be related to the function arguments due to non-determinism. We further ignore the condition in the formula as it has no further relevance either way.

$$(\text{true} \parallel (x = X \wedge y = Y \wedge z = Z)_0^3 \vee (x = X \wedge y = Y \wedge z = Z)_1^9)$$

If the analysis chooses w.l.o.g. to first proceed with post-condition 1, it will abduce that `X.data` needs to be allocated and share this information with the rest of the world:

$$(\textcolor{red}{X.data} \mapsto \ell_1 \parallel (x = X \wedge y = Y \wedge z = Z : \textcolor{red}{X.data} \mapsto \ell_1)_0^3$$

$$\vee (x = X \wedge y = Y \wedge z = Z \wedge \textcolor{red}{return}_{\text{nested}} = \ell_1 : \textcolor{red}{X.data} \mapsto \ell_1)_1^{11})$$

Thus, the current post-condition in the `then` branch now also requires as a pre-condition that `X.data` is allocated and will not be unsound due to missing this information. The analysis can then choose to proceed with the current post-condition 0 and find that it can relate the branching condition with the function arguments. Therefore, the world needs to be split, as the two cases of condition are expressed as part of the world's pre-condition. To be more precise, the world is split based on whether `Y` is `NULL` (with omitted anchor equalities):

$$\begin{aligned}
& (Y \neq \text{NULL} : X.\text{data} \mapsto \ell_1 \parallel (\cdots \wedge Y \neq \text{NULL} : X.\text{data} \mapsto \ell_1)_0^4 \\
& \quad \vee (\cdots \wedge \text{return}_{\text{nested}} = \ell_1 \wedge Y \neq \text{NULL} : X.\text{data} \mapsto \ell_1)_1^{11})_0, \\
& (Y = \text{NULL} : X.\text{data} \mapsto \ell_1 \parallel (\cdots \wedge Y = \text{NULL} : X.\text{data} \mapsto \ell_1)_0^6 \\
& \quad \vee (\cdots \wedge \text{return}_{\text{nested}} = \ell_1 \wedge Y = \text{NULL} : X.\text{data} \mapsto \ell_1)_1^{11})_1.
\end{aligned}$$

The two worlds will then abduce different required pre-conditions in further steps and finally result in the following (simplified) contracts for the function `nested`:

$$\begin{aligned}
& (y \neq \text{NULL} : x.\text{data} \mapsto \ell_1 * y.\text{data} \mapsto \ell_2 \parallel \\
& \quad (y \neq \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_2 : x.\text{data} \mapsto \ell_1 * y.\text{data} \mapsto \ell_2) \\
& \quad \vee (y \neq \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_1 : x.\text{data} \mapsto \ell_1 * y.\text{data} \mapsto \ell_2)), \\
& (y = \text{NULL} : x.\text{data} \mapsto \ell_1 * z.\text{data} \mapsto \ell_3 \parallel \\
& \quad (y = \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_3 : x.\text{data} \mapsto \ell_1 * z.\text{data} \mapsto \ell_3) \\
& \quad \vee (y = \text{NULL} \wedge \text{return}_{\text{nested}} = \ell_1 : x.\text{data} \mapsto \ell_1 * z.\text{data} \mapsto \ell_3)).
\end{aligned}$$

4.1 Comparison with Disjunctive Domains

It may be tempting to consider shared abduction with worlds to be just a disjunctive closure of conjunctive formulas used commonly in various abstract interpretation approaches. However, when using a disjunctive closure, the symbolic execution is typically performed independently for each disjunct, perhaps followed by attempts to join some of the disjuncts or to prune them away using entailment checks—as done in [10,6,13]. In contrast, our analysis with worlds differs in that (1) the worlds are, in fact, not purely disjunctive due to a single precondition shared by all current post-conditions in a world and due to working with sets of worlds, (2) state splits either result in two new post-conditions or two new worlds, and (3) the symbolic execution from a single disjunct can influence all other disjuncts in the same world via shared abduction.

5 Biabductive Loop Acceleration via Shape Extrapolation

We first introduce the central steps of our technique for a simplified setting. In this setting, loops only have loop conditions of the form `x ≠ NULL` where `x` is a function parameter. Furthermore, we assume that loops do not contain branching. We will show how to lift these restrictions in Section 5.2.

```

void free_list(node *x) {
  while (x != NULL) {
    node *aux = x;
    x = x->next;
    free(aux);
  }
}

```

Listing 5: Deallocating a list

We will explain the steps of our biabductive loop acceleration with the help of the example in Listing 5, which falls into the fragment of programs allowed in the simplified setting. The example shows a simple loop that frees a given list node by node. As such, the expected contract would be $(ls(x, NULL) \parallel \text{emp})$.

5.1 Basic Biabductive Loop Acceleration

Whereas Procedure 1 describes biabductive loop acceleration on a high level, the following paragraphs describe the main steps of the procedure in more detail.

Our algorithm first analyzes a single loop iteration starting from the initial analysis state s_0 . If this analysis run ends in a state s_1 , the algorithm then continues by determining which parts of s_1 describe the shape of the traversed data structure, i.e., the traversed singly-linked lists in our simplified setting. To this end, the algorithm partitions the candidate precondition as well as the current post-condition of the state s_1 into subformulas $\tau_{pre/curr}$ and $\rho_{pre/curr}$ such

that the τ formulas contain the *transformed*¹⁰ parts of the state that should be related to the shape, whereas the *remaining* parts of the state are collected in the subformulas ρ , which comprise both completely unchanged predicates as well as changed memory locations that are not part of the shape. This separation is done for both the pre- and current post-condition of the state s_1 to capture changes to the shape of the data structure. Some more technical details of the partitioning, which are not needed now, will be presented in Section 5.3.

Example 3. For Listing 5, the analysis finds the state $s_1 := (X.\text{next} \mapsto \ell_1 \parallel x = \ell_1)$ after one loop iteration. There, the partition of s_1 is trivially $\tau_{pre} := s_1.pre$ and $\tau_{curr} := s_1.curr$ as this simple loop does not affect anything except the traversed list. On the other hand, the inner loop of Listing 2 does not change the shape of the traversed list but accesses and changes further parts of the program state. As a result, the partitions are $\tau_{pre} = \tau_{curr} := I.\text{next} \mapsto \ell_1 * I.\text{elem} \mapsto \ell_2$ and $i = I$ or $i = \ell_1$, respectively, for the predicates that relate to the list and $\rho_{pre} = o.\text{wgt} \mapsto \ell_3 * \text{sum} \mapsto \ell_4$ and $\rho_{curr} = o.\text{wgt} \mapsto \ell_3 * \text{sum} \mapsto (\ell_4 + \ell_3 \cdot \ell_2)$ for the ones relating to the context.

The main step of our procedure is the construction of the candidate loop invariant s_{inv} . For that, we first need to find an abstract description of the shape of the involved data structures. The abstraction must satisfy specific properties described below that are needed to ensure the soundness of the approach. We call this step *shape extrapolation* and provide a minimum viable heuristic implementing it in Section 5.3. However, we stress that this algorithm can be changed as long as the properties in Figure 4 hold.

Procedure 1 Biabductive loop acceleration

Input: A function f consisting of a loop l with body $body_l$ and exit condition e_l

Output: A sound contract c for f or FAILURE

$s_0.pre \leftarrow \text{true}, \quad s_0.curr \leftarrow \bigwedge \{x = X \mid x \in PVar\}$

Compute $s_1 \leftarrow A_{B,\alpha}(body_l, s_0)$

$(\tau_{pre} * \rho_{pre} \parallel \tau_{curr} * \rho_{curr}) \leftarrow \text{PARTITION}(s_1)$

$\mathcal{P}, \mathcal{Q} \leftarrow \text{SHAPEEXTRAPOLATION}(\tau_{pre}, \tau_{curr})$

Construct s_{inv} from $\mathcal{P}, \mathcal{Q}, \rho_{pre}$, and ρ_{curr}

$s_2 \leftarrow A_{B,\alpha}(body_l, s_{inv})$

Check that $s_2.curr \vdash s_{inv}.curr$

Construct s_{final} from $\mathcal{P}, \mathcal{Q}, \rho_{pre}$, and ρ_{curr}

return $c \leftarrow s_{final}$

¹⁰ Here “transformed” means the changed value of the loop variable x and the shape that is described in between the old and the new value of x .

In general, we need shape extrapolation to abstract the two subformulas τ_{pre} and τ_{curr} to list-segment shapes $\mathcal{P}(X, x)$ and $\mathcal{Q}(X, x)$ where the parameter X

denotes the first node of the list segment and x denotes the current position in the segment; we omit the parameters X and x and simply write $\mathcal{P}$ and $\mathcal{Q}$ when there is no danger of confusion. We require $\mathcal{P}$ and $\mathcal{Q}$ to satisfy the three properties given in Figure 4. These conditions are generalized in Figure 5, and the proof of soundness (see [29, Appendix D.2]) shows that they allow to establish a loop invariant.

Intuitively, Point (1) of Figure 4 simply ensures that $\mathcal{P}$ and $\mathcal{Q}$ are actual abstractions of τ_{pre} and τ_{curr} , respectively. In addition, Property (2) ensures that $\mathcal{Q}$ only describes the so-far traversed and transformed part of the list. Thereby, if $X = x$, i.e., at the start of the loop, the so-far traversed and transformed part of the list must be empty. Finally, (3) states that consecutive list segments may always be composed into one list segment.

1. $\tau_{pre} \vdash \mathcal{P}$ and $\tau_{curr} \vdash \mathcal{Q}$,
2. $\mathcal{Q} \wedge X = x \vdash \text{emp}$,
3. $\mathcal{P}[a/X, b/x] * \mathcal{P}[b/X, c/x] \vdash \mathcal{P}[a/X, c/x]$

Fig. 4: Extrapolation properties

Example 4. In the example in Listing 5, the shape τ_{pre} obtained after one loop iteration is extrapolated (see Procedure 2) to the formula $ls(X, \ell_1)$. Since $x = \ell_1$, the shape $\mathcal{P}$ becomes $ls(X, x)$. On the other hand, $\tau_{curr} = \text{emp}$ does not contain any spatial predicates, and so the extrapolation produces emp as $\mathcal{Q}$, since the transformation of the list consists in deleting it – if the list was just traversed, we would obtain $ls(X, x)$. It is easy to verify that all properties of Figure 4 are satisfied.

In contrast to previous analyses, we explicitly construct a candidate loop invariant from the abstract shapes $\mathcal{P}$ and $\mathcal{Q}$ and subsequently check whether it is sound. The candidate loop invariant s_{inv} is meant to describe an intermediate state of the loop:

$$s_{inv} := (\rho_{pre} * \mathcal{P} * \mathcal{P}[x/X, NULL/x] \parallel \rho_{curr} * \mathcal{Q} * \mathcal{P}[x/X, NULL/x]).$$

The pre-condition of this state contains two (sub-)shapes $\mathcal{P}$ and $\mathcal{P}[x/X, NULL/x]$ where the first describes the already traversed list segment starting in X and ending in the current value of x , whereas the latter denotes the not yet traversed part of the list starting at x and ending in $NULL$. Due to the extrapolation Property (3), the two sub-shapes together form the full extrapolated shape $ls(X, NULL)$. In contrast, the post-condition also takes into account the effects of the loop on the already traversed list segment and, thus, contains $\mathcal{Q}$ instead of $\mathcal{P}$.

To prove that s_{inv} is a loop invariant, the analysis also needs to check whether the post-condition's memory footprint is sufficient for another loop iteration and whether it also holds after this iteration. This is proven by analyzing another loop iteration starting from s_{inv} in which the abduction of new pre-condition predicates is disallowed, thus forcing the analysis to fail if the shapes describe an insufficient memory footprint.

Suppose the invariant checking step successfully finishes the symbolic execution of the loop body in some state s_2 . In that case, this implies that the loop body can be safely executed from the state s_{inv} . Next, we check whether $s_2.curr \vdash s_{inv}.curr$, i.e. whether s_{inv} is actually a loop invariant. If the check succeeds, the shapes are sound for all loop iterations, and the loop acceleration procedure can continue with the final step.

Example 5. It trivially holds that the following state is invariant for the loop in Listing 5, i.e., it is sound before and after each loop iteration.

$$s_{inv} := (ls(X, \ell_1) * ls(\ell_1, NULL) \parallel x = \ell_1 : ls(\ell_1, NULL))$$

As the state depicts the program at an arbitrary point of the iteration, it contains both the already traversed shape $ls(X, \ell_1)$ in the pre-condition (which has been freed in the post-condition) and the unchanged, still-to-traverse shape $ls(\ell_1, NULL)$.

Finalizing. Lastly, the loop analysis is finalized by constructing the final state reached after finishing the loop from the shapes $\mathcal{P}$ and $\mathcal{Q}$ as

$$s_{final} := (\rho_{pre} * \mathcal{P}[NULL/x] \parallel x = NULL * \rho_{curr} * \mathcal{Q} * \mathcal{P}[x/X, NULL/x])$$

This state is simply obtained from s_{inv} by adding the negated loop condition $x = NULL$ and using extrapolation Property (3) to simplify the pre-condition. If the extrapolated shape additionally satisfies the property $\mathcal{P}(x/X, NULL/x) \wedge x = NULL \vdash \text{emp}$ (we call this Property (3.5)), which is natural for traversing linked lists until the end, the final state can be simplified even further.

Example 6. For the loop in Listing 5, the freed list is represented by the shape $\mathcal{Q}$ being empty, making the final state $s_{final} := (ls(X, NULL) \parallel x = NULL * ls(x, NULL))$. Since the list segment to $NULL$ satisfies Property (3.5), we obtain the expected final state. In addition, this state is also the contract of the function `free_list`, and thus the analysis reaches its end for this function.

5.2 Lifting Restrictions on Biabductive Loop Acceleration

We now explore how the restrictions introduced above can be lifted to make biabductive loop acceleration more applicable in practice. We write $\bar{x}$ for an ordered list of elements x_i with $0 \leq i \leq n$ for some n . We denote by $f(\bar{x})$ the ordered list $\bar{y}$ where $y_i = f(x_i)$ for $0 \leq i \leq n$.

```
void either_list(node *x) {
  node *head = x;
  while (x != NULL
    && x->next != head)
    {...}
}
```

Listing 6: Cyclic/to-null lists

Extension: General Loop Conditions. The first restriction that we lift concerns the loop condition. We assume that the loop condition e_l is of the form $e_l = \bigwedge_i e_i$ with single atomic conjuncts e_i of arbitrary form. An example of such a loop can be seen in Listing 6, which handles both cyclic and null-terminated lists equally. Handling such a more general loop condition requires further adjustments to the loop acceleration procedure. To be able to express multiple exit conditions that relate to multiple different variables, the algorithm needs to be able to describe the traversed shape relative to these variables. Hence, the shapes $\mathcal{P}$ and $\mathcal{Q}$ are now parameterized over all program variables changed throughout the loop—namely, all variables x for which $s_1.curr.H \not\vdash x = X$. We call the set of these variables χ and re-define the $\mathcal{P}$ and $\mathcal{Q}$ shapes as $\mathcal{P}(\bar{X}, \bar{x})$ and $\mathcal{Q}(\bar{X}, \bar{x})$, respectively, where $\bar{x}$ is the ordered list of the variables from χ that occur in $\mathcal{P}$ and $\mathcal{Q}$, and $\bar{X}$ is the ordered list of the corresponding anchor variables. Below, we will use $\mathcal{P}(\bar{a}, \bar{b})$ to

handles both cyclic and null-terminated lists equally. Handling such a more general loop condition requires further adjustments to the loop acceleration procedure. To be able to express multiple exit conditions that relate to multiple different variables, the algorithm needs to be able to describe the traversed shape relative to these variables. Hence, the shapes $\mathcal{P}$ and $\mathcal{Q}$ are now parameterized over all program variables changed throughout the loop—namely, all variables x for which $s_1.curr.H \not\vdash x = X$. We call the set of these variables χ and re-define the $\mathcal{P}$ and $\mathcal{Q}$ shapes as $\mathcal{P}(\bar{X}, \bar{x})$ and $\mathcal{Q}(\bar{X}, \bar{x})$, respectively, where $\bar{x}$ is the ordered list of the variables from χ that occur in $\mathcal{P}$ and $\mathcal{Q}$, and $\bar{X}$ is the ordered list of the corresponding anchor variables. Below, we will use $\mathcal{P}(\bar{a}, \bar{b})$ to

denote the predicate $\mathcal{P}(\overline{X}, \overline{x})[\overline{a}/\overline{X}, \overline{b}/\overline{x}]$, i.e., the predicate obtained from $\mathcal{P}(\overline{X}, \overline{x})$ by simultaneously substituting the variables $\overline{X}$ with $\overline{a}$, and $\overline{x}$ with $\overline{b}$ (we will use the same notation for $\mathcal{Q}$). Note that for lists, this is equal to setting the two parameters of the list segment predicate to a and b , respectively.

With this notation, we re-define the properties of extrapolation to consider the new parameters in Figure 5. We further define a mapping $\text{EXIT}(x)$ of variables $x \in \chi$ to the values they can have at a loop exit. As these values can be challenging to determine from the loop condition alone, we restrict the map to hold only logically constant values, i.e., NULL or other program variables outside of χ (as their values stay constant throughout the loop), and define the other entries to map to fresh logical variables instead.

Furthermore, the post-condition of the final state has to encode that any of the loop conditions can be unsatisfied for the program to leave the loop. This is done by taking the disjunction of the previous final state post-condition combined with one dissatisfied loop condition (note that the disjunction represents the world's current post-conditions):

$$s_{\text{final}}.curr := \bigvee_i (\neg e_i * \rho_{curr} * \mathcal{Q}(\overline{X}, \overline{x}) * \mathcal{P}(\overline{x}, \text{EXIT}(\overline{x}))).$$

1. $\tau_{pre} \vdash \mathcal{P}(\overline{X}, \overline{x}) \wedge \tau_{curr} \vdash \mathcal{Q}(\overline{X}, \overline{x})$,
2. $\mathcal{Q}(\overline{X}, \overline{x}) \wedge \bigwedge_{x \in \chi} X = x \vdash \text{emp}$,
3. $\mathcal{P}(\overline{a}, \overline{b}) * \mathcal{P}(\overline{b}, \overline{c}) \vdash \mathcal{P}(\overline{a}, \overline{c})$.

Fig. 5: General extrapolation properties

Extension: Branching Loop Body. Branching in loop bodies can be handled by collecting all states s_1 after the first loop iteration analysis, extrapolating their shapes, and combining them if possible into one compound shape via a join operation akin to the ones described in [10] or [26]. Because such an operation is mostly orthogonal to the central ideas of shape extrapolation, we refer to the literature for more details.

Extension: Overlapping Shape Changes. The extension to allow for more general loop conditions can lead to problems with shape extrapolation if the involved shapes overlap, i.e., if the new and old memory locations to which program variables point to are the same. This can, e.g., happen if a list is reversed (see [29, Appendix F.2]). Such cases can be detected if the new value of a program variable in χ is the anchor of another variable. In the example of list reversal, the program variable tracking the reversed list will be set to the initial value of the original list as that list's first node becomes the last node in the reversed one. Such an overlap would cause problems in the implementation of SHAPEEXTRAPOLATION presented as Procedure 2. To circumvent this problem, the analysis symbolically executes further additional loop iterations to find a program state in which there is no overlap anymore, and only then performs the extrapolation.

Extension: Further Loop Effects. As depicted in Listing 2, loops can not only traverse data structures but also change the program state in arbitrary ways. In such cases, the candidate invariant s_{inv} might not be an actual invariant, i.e. $s_2.curr \not\vdash s_{inv}.curr$. To handle such cases, we apply a join in the corresponding pure value domain of the analysis. In the simplest case, this step exchanges the values of variables and memory locations that are the cause of $s_2 \not\vdash s_{inv}$ with the value $\top$. In the example Listing 2, the value stored at `sum` after the first iteration is $\ell_{sum} + \ell_1 \cdot \ell_w$, resulting in the points-to predicate $sum \mapsto \ell_{sum} + \ell_1 \cdot \ell_w$ being a part of s_{inv} . After the second iteration, the

predicate changes to $sum \mapsto \ell_{sum} + \ell_1 \cdot \ell_w + \ell_2 \cdot \ell_w$, which does not entail its counterpart in s_{inv} . However, by joining the two values of the memory location to $sum \mapsto \top$, the entailment is ensured. The same problem actually occurs if the initial value of a variable does not entail its representation in the invariant, e.g., because it is set to a constant in the loop (see [29, Appendix F.4]). In this case, we also need to abstract the variable's values in $s_{inv}.curr$ to $\top$, thus guaranteeing that s_{inv} also holds before the first iteration.

Theorem 2 (Soundness of Shape Extrapolation). *Let $A_{B,\alpha}$ compute only sound contracts for loop-free functions. If Procedure 1 uses $A_{B,\alpha}$, then Procedure 1 with all extensions described in this section applied to a loop l either fails or returns a contract (P, Q) such that $\{P\} l \{Q\}$. **Proof:** See [29, Appendix D.2].*

5.3 Shape Extrapolation

We now propose a concrete shape extrapolation procedure based on the principles presented above. This procedure is supposed to be the easiest possible heuristic that suffices to find reasonable loop invariants. To this end, it follows the original idea of obtaining inductive shapes through abstraction, but in a “smarter” way.

The initial partitioning is one of the most crucial steps for our shape extrapolation procedure. The τ formulas are built by collecting all predicates that describe the shape traversed, i.e., the shape between the anchors and the new values of the variables in χ . These shapes contain all transitively reachable predicates, where reachability is defined spatially. Thereby, a predicate is reachable if there exists a sequence of points-to and list predicates that pairwise overlap in their source/drain variables, modulo variable equalities. For example, if $x \in \chi$, then $\ell_1 = \ell_4 : X \mapsto \ell_1 * ls(\ell_1, \ell_2) * \ell_4.data \mapsto \ell_3$ contains only predicates reachable from the anchor X .

In addition to the partitioning of variables, our concrete shape extrapolation algorithm also needs to know the new value of the variables in χ . We encapsulate this information in the *transformation map* μ which maps $x \in \chi$ to $\ell_x \in LVar$ such that $s_1.curr.\Pi_P \vdash x = \ell_x$. Recall that, due to the normal form of SL , every program variable only occurs in a single equality such as $x = \ell_1$, and so μ can be computed by simply comparing their values before and after the loop.

Procedure 2 gives a detailed description of our concrete shape extrapolation procedure. It computes $\mathcal{P}$ (and $\mathcal{Q}$) by first extrapolating the corresponding τ_i into two copies τ_i^1 and τ_i^2 . These two copies are supposed to represent the shape accessed by two consecutive loop iterations via an intermediate, fresh auxiliary location. The resulting formulas are then combined via separating conjunctions and abstracted by the abstraction function α to form the abstract shapes θ , which

Procedure 2 ShapeExtrapolation

Input: τ_{pre}, τ_{curr}

Output: Parametric extrapolated shapes $\mathcal{P}$ and $\mathcal{Q}$

Compute the maps χ and μ as described in the text

// $i \in \{pre, curr\}$, x' fresh

$\tau_i^1 \leftarrow \tau_i[x'/\mu(x) \mid x \in \chi]$

$\tau_i^2 \leftarrow \tau_i[x'/X \mid x \in \chi]$

$\theta_i \leftarrow \alpha(\tau_i^1 * \tau_i^2)$

$\mathcal{P}(\bar{x}_1, \bar{x}_2) \leftarrow \theta_{pre}[\bar{x}_1/\bar{X}, \bar{x}_2/\mu(\bar{x})]$

$\mathcal{Q}(\bar{x}_1, \bar{x}_2) \leftarrow \theta_{curr}[\bar{x}_1/\bar{X}, \bar{x}_2/\mu(\bar{x})]$

return $\mathcal{P}, \mathcal{Q}$

in turn get parameterized by renaming schemas to the final $\mathcal{P}$ and $\mathcal{Q}$. The use of two copies is a heuristic that has proven to be reliable in making the abstraction find better abstract shapes. Note that, in Procedure 2, we omit additional renamings of logical

variables for clarity, as these only help to guide the abstraction but do not affect the resulting shapes any further.

Example 7. In the example from Listing 2, the inner loop can be extrapolated as follows: The procedure takes the effect $\tau_{pre} := I.\text{elem} \mapsto \ell_1 * I.\text{next} \mapsto \ell_2$ from the transformation and introduces the two auxiliary formulas $\tau_{pre}^1 := I.\text{elem} \mapsto \ell_1 * I.\text{next} \mapsto i'$ and $\tau_{pre}^2 := i'.\text{elem} \mapsto \ell_1 * i'.\text{next} \mapsto \ell_2$ where i' is the auxiliary location representing the intermediate value of i . From these formulas, the abstraction then finds the abstract shape $\theta_{pre} := ls(I, i)$. This abstracted shape is then the basis for the extrapolated shape $\mathcal{P}$. Similarly, $\mathcal{Q}$ is computed to be $ls(I, i)$, too.

5.4 Limitations

We note that shape extrapolation is a heuristic, which is sound (see Theorem 2) but inherently incomplete. The extrapolation step can fail if the partitioned information does not suffice to find a reasonable shape, e.g., if the abstraction function cannot find a fitting inductive shape predicate. However, since our shape extrapolation procedure imitates the loop acceleration procedure of the original analysis, it is guaranteed to be applicable for at least the same programs but in a fundamentally sound (and oftentimes faster) way. Furthermore, we note that shape extrapolation is currently limited to list-like data structures that are traversed linearly. List manipulation is, however, by far the most frequent data structure pattern in low-level code, and so we have focused our efforts on this kind of data structures, in accordance with prior work [6, 13]. Nonetheless, we believe that shape extrapolation, which is based on the intuition of locality, can be extended toward tree-like data structures in future work.

6 Implementation and Experimental Evaluation

6.1 Prototype Implementation

We have implemented our techniques as a proof-of-concept in the prototype analyzer Broom [13] written in OCaml and call the resulting tool Brush. It is available as an artifact on Zenodo [28]. The original Broom implements a biabduction-based shape analysis with a focus on low-level primitives and byte-precise memory management and is sound for functions without branching [13, see Theorem 3]. However, since Broom is also still a prototype that focuses more on exact handling of complex memory manipulation than on scalability, neither Broom nor Brush are able to handle large-scale code bases yet. Our new techniques, especially shape extrapolation, improve scalability, but Brush still shares most of its code with Broom and is thus not as mature as industrial-strength tools such as Infer.

6.2 Implementation Limitations

As Brush is largely based on the source code of Broom and does not differ much from it apart from our new techniques, they share mostly the same limitations. On the one hand,

neither tool supports recursive functions. Similarly, they can handle neither stack allocations nor switch-case statements. On the other hand, the logic both tools are based on contains only inductive predicates for linked lists with parameters describing the shape of single nodes. Therefore, the tools can, in general, not analyze programs containing other inductive data structures.

Furthermore, we remark that the running times of Broom and Brush are much higher than for comparable tools, which in part is due to the need for precise pointer arithmetic. This precision is achieved, among other things, by calling an SMT solver, which is more costly for simpler cases than using native solvers, such as in Infer.

6.3 Case Study

We have conducted experiments with two research questions in mind: (1) whether Brush can handle new use cases that existing tools cannot handle; (2) whether Brush is also at least as efficient as Broom or would even improve scalability.

Qualitative Experiments. To answer research question (1), we ran all four analyzers on selected examples that are either presented in [13, Table 1], are a part of the test suite for Broom, or are hand-crafted test cases for shared abduction and shape extrapolation. The results of our case study are depicted in Table 1. All test files are included in the accompanying material. We primarily investigated whether the analyzers found the expected bugs and sound contracts or whether they report other spurious errors.

We note that the biabduction-based shape analysis that Infer was based on is deprecated nowadays, and Infer’s focus has shifted from over-approximation to under-approximation (see [23,18]). Due to this, we not only compare Brush with the release v1.1.0¹¹ which was also used for comparison in [13], but also with its predecessor tool Abductor.¹² We excluded the second-order biabduction tool S2 from our experiments since it is quite limited and cannot be applied to most of our benchmark programs.¹³

The ten tests from [13, Table 1] are program fragments of 30–200 LOC. Each test consists of a set of library functions, including creation of a linked-list, insertion and deletion of an element from the list. Moreover, 8 of the tests also contain a top-level test harness performing a concrete manipulation of the particular list. There are three types of lists: (i) circular doubly-linked lists, (ii) linux-lists taken from the Linux kernel, and (iii) intrusive lists.¹⁴ The 43 tests from `broom/tests` are regression tests of Broom. Each one is usually up to 10 LOC and tests the analysis of a particular kind of statement.

¹¹ Available at <https://github.com/facebook/infer/releases/tag/v1.1.0>.

¹² Available at <http://www0.cs.ucl.ac.uk/staff/p.ohearn/abductor.html>.

¹³ Of the 73 programs in Table 1, the tool reported internal errors for 56 cases while causing segmentation faults for nine further cases. If we only compare the 52 programs without loops, it fails for 43 instances and causes segmentation faults in 4 further cases. The internal errors range from unsupported language features such as pointer arithmetic (3 programs) to linker errors with unknown symbols (13 cases/6 cases without loops) and unsupported type casts (35/30 cases). All of these cases are correctly handled and accepted by standard C compilers as utilized as frontends by Broom and Brush.

¹⁴ Described by [31] and implemented in <https://github.com/robbiev/coh-linkedlist>.

Table 1: Examples handled correctly (✓) and incorrectly (×) by the analyzers

Class of inputs	# of test cases	Broom	Infer	Abductor	Brush
[13, table 1]	10	10✓/0×	0✓/10×	0✓/10×	10✓/0×
tests from broom/tests	47	47✓/0×	34✓/14×	12✓/35×	47✓/0×
*_branches.c	2	0✓/2×	0✓/2×	0✓/2×	2✓/0×
nested_*.c	3	1✓/2×	3✓/0×	3✓/0×	3✓/0×
motivation*.c	3	0✓/3×	0✓/3×	0✓/3×	3✓/0×
sll*.c	3	3✓/0×	0✓/3×	0✓/3×	3✓/0×
other	5	3✓/2×	0✓/5×	3✓/2×	5✓/0×
overall	73	64✓/9×	47✓/26×	18✓/55×	73✓/0×

The newly added hand-crafted use cases are small-scale programs (10–70 LOC), which are, however, rather challenging for the existing analyzers. In particular, the *_branches test cases contain multiple cases of non-determinable branching, which can lead to the unsoundness described in Section 2.1. The nested_* programs contain multiple examples of nested loops and nested lists. The sll_* test cases contain whole programs that create, iterate and destroy singly-linked lists. The motivation programs are as described in Section 2. Lastly, the other test cases contain programs with more complex list allocation, deallocation, and transformation.

We specifically note that we have not used common benchmark sets such as the SV-COMP memory-safety benchmark, as these consist primarily of closed programs and focus on data structures that cannot be described by the logic of Broom and Brush. Thus, these benchmark sets lie outside the scope of this work, and we have instead used test cases that allow us to evaluate our research questions explicitly.

In the table, we use ✓ to denote that at least one (sound) contract was computed for each function within the particular example and that the expected errors were reported without false positives. On the other hand, we use × to denote that either no contract could be computed (for a function that would have a sound contract) or the respective tool reported a false positive.

We conclude that shared abduction and shape extrapolation enable Brush to work for strictly more programs than Broom or any of the other tools. This is especially important, as these small but challenging test cases are mostly based on realistic iteration patterns that can be found in code bases such as the Linux kernel.

Quantitative Experiments. In another series of experiments, we also evaluated the runtime of Brush versus Broom¹⁵ on test cases that both tools can handle. Some test cases were split to survey the runtime of single, interesting functions without their calling context. All experiments were run ten times on an Intel Core i7-1260P CPU with 32GiB RAM, and we took the mean over all runs. A plot of the results can be found in Figure 6, the raw data is displayed in [29, Appendix E]. The overall means for programs with loops are 1.85s for Brush to 38.2s for Broom, whereas the means for loop-free branching programs are 22.3s for Brush and 24.8s for Broom. The overall means are 7.24s for Brush to 14.0s for Broom. These numbers show that Brush provides a signif-

¹⁵ We used the commit a361d01badf45c420b57158f2e6d738cb45d1dd9 found at <https://pajda.fit.vutbr.cz/rogalew/broom> with small additional bug fixes.

Table 2: Loop iterations analyzed by the analyzers (Brush/Broom) until a fixed point was reached, as well as how many loops were present in the programs (in brackets)

copy_alloc	dll-as-sll-traverse	reversal	nested_lists2
2/6 (1)	2/6 (1)	3/8 (1)	6/14 (3)
sll-fst-shared	sll-fst-shared-alloc	sll-fst-shared-iter	sll
4/12 (2)	2/8 (1)	2/6 (1)	6/20 (3)
sll-alloc	sll-shared-sll-after	sll-shared-sll-after-alloc	overall
2/6 (1)	6/21 (3)	4/14 (2)	39/121 (19)

icant speedup over Broom. For the examples with branching but without loops, we at least find a tendency towards faster runtime for Brush.

We directly relate the time improvements of Brush with the number loop iterations analyzed. As seen in Table 2, Brush only requires a fraction of loop iterations due to shape extrapolation for all examples from Table 1, that both Broom and Brush can handle and which contain loops. The mean over all examples is 3.55 iterations for Brush and 11 for Broom. We note that in most cases, Brush requires exactly two iterations per loop, which corresponds to the initial analysis and the invariant checking iteration as described in Section 5. Only the case *reversal* (see [29, Appendix F.2]) requires 3 iterations, as it reverses a list such that the new and old shapes partially overlap. As described in the paragraph about *Extension: Overlapping Shape Changes*, the analysis requires an additional iteration to reach a state in which this overlap has been removed.

We conclude that shared abduction improves both the precision and the performance of the analysis compared to Broom, while biabductive loop acceleration considerably improves the performance of the analysis for the benchmarks.

7 Related Work

Biabduction-based Shape Analysis. Our work builds on biabduction-based shape analysis introduced by Calcagno et al. [6,5] and later implemented in Infer [4]. Our new techniques avoid the unsoundness issues of the first phase of the analysis and constitute significant theoretical and practical advancements as demonstrated in Sections 4 to 6.

The approach of [6,5] was extended in the Broom analyzer [13] by ways of handling low-level primitives and byte-precise memory handling. Since these extensions are orthogonal to the problems of unsoundness, our techniques are equally effective for improving Broom as depicted in Section 6.

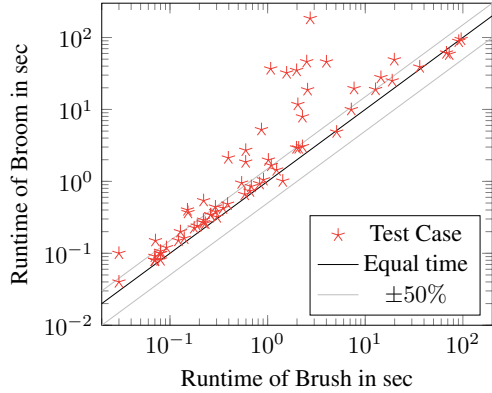


Fig. 6: Runtime of Broom and Brush in sec. for tests from Table 1 that both tools handled

Another related work is the *second-order biabduction* by Le et al. [17]. Their approach does not consider a fixed class of inductive predicates, but discovers them as part of the analysis by instantiating second-order variables with a technique called *shape inference*. The analysis first collects the unknown predicates with corresponding relational assumptions and synthesizes fitting shapes in a second step. Their computation method ensures that these shapes make the resulting contracts sound. Albeit this makes their technique similar to shape extrapolation, the approach of [17] can handle more complex shapes of dynamic data structures. On the other hand, it requires solving the complex predicate inference problem for which their tool uses a relatively simple algorithm. In our experience, this algorithm can easily fail even for programs with simple data structures if they are not compatible with the shape inference procedure. Furthermore, their implementation is rather limited, as described in Section 6.

Very recently, Spies et al. [30] combined biabductive reasoning with auto-active, foundational program verification. Their tool Quiver takes C programs and specification sketches as annotations as input, translates them into a representation in the Caesium C semantics [27], and finally infers and proves full function specifications in the proof assistant Coq. The central reasoning mechanism of Quiver is called *abductive deductive verification*, which is closely related to biabduction. As Quiver does not only work with predicates for memory safety but also with a refinement type system for C, it has a broader focus than our work. However, Quiver requires the user to provide specification sketches, refinement types, and loop invariants, while our work focuses on fully automated, biabductive shape analysis.

Other Shape Analyses. There are many different shape analysis methods not based on biabduction in literature. Of these, the Predator analyzer [10] based on *symbolic memory graphs* is quite successful with regard to the Competition on Software Verification (SV-COMP), see [2,3]. Their approach handles abstraction, entailment, and state pruning as special cases of a general graph joining procedure. They focus on closed programs, and their approach uses function summaries that are computed in a top-down fashion, following the call tree (whereas biabduction-based shape analysis works from the bottom up). This top-down fashion requires a re-analysis of functions for different contexts but can ignore irrelevant code paths. As Predator implements a classic forward analysis that does not compute contracts with pre-conditions, Predator circumvents the problem of unsoundness. On the other hand, Predator only works on closed programs and can thus not be used for modular and incremental analysis.

Another approach to shape analysis has been recently introduced by Illous et al. [15,14]. They utilize *transformers* to describe the effects of functions and compute these with regard to the calling context in a top-down fashion, as in Predator. The transformer-based analysis is built around a transformer abstract domain and is based on abstract interpretation. We note that the use of transformers has partially motivated the inner workings of our shape analysis procedure. The authors noted that biabduction might be applicable for transformers as well, and we strengthen this point by showing how our shape extrapolation procedure combines both ideas to some degree.

We also like to mention the work [9], where the authors consider overlaid data structures. Their technique is based on a fragment of separation logic that differentiates per

object and per field separation. Note that the separation logic fragment we base our work on uses per field separation, while per-object separation is only enforced implicitly.

Other Related Analyses. Recent work about *Incorrectness (Separation) Logic* (ISL) [23,24,18] has introduced a different approach to program analysis. This line of work does not focus on verifying the absence of memory bugs in an over-approximating way but instead tries to find bugs in an under-approximative way. Although the bug-finding ability of incorrectness logics makes them very useful in practice, over-approximating analyses are still relevant for certification and low-level systems software.

Lastly, the emergence of incorrectness logic has also motivated the development of combined logic systems that inherit the benefits of both over- and under-approximating logics. Recent work in this direction includes *Exact Separation Logic* [20] as well as *Outcome (Separation) Logic* [32,33]. In particular, *tri-abduction*, introduced in [33], offers an alternative solution to the branching problem we address with shared abduction. By solving the abduction problem for the pre-conditions required by both branches, the tri-abduction operation can potentially compute better contracts than the (greedy) techniques proposed in this paper, which will first solve the abduction problem for one branch and then for the other. This increased precision, however, comes with the burden of implementing a new operator, whereas we can simply lift existing implementations of biabduction operators to shared abduction. We believe that the increased precision of the triabduction operation is rarely needed, and thus the more lightweight solution of shared abduction is preferable in practice. For our experiments, the precision of shared abduction was sufficient. At the same time, the approach from [33] has not yet been implemented into a tool that could be used for experimental comparison.

8 Conclusion and Future Work

This work introduces the two novel techniques of shared abduction and biabductive loop acceleration with shape extrapolation. We provide soundness proofs for both techniques and implement them in our prototype analyzer Brush, which is based on the state-of-the-art analyzer Broom. We experimentally demonstrate that these techniques enable our biabduction-based shape analysis to find sound contracts in a single analysis phase. In particular, we show that shared abduction and shape extrapolation enable Brush to analyze strictly more programs than Broom or any of the Infer versions, and to considerably improve the performance compared to Broom.

While our work is limited to non-recursive programs, we believe that shape extrapolation can easily be extended to recursive programs (e.g., tree traversals) and that this direction constitutes an exciting avenue for future work. We also hope to incorporate techniques that track the content of data structures, i.e., the data values stored in the data structure. Specifically, we would like to enrich the logic and biabduction procedure to track data values, e.g., for tracking the value of `sum` in the example of Listing 2. For this, we plan to take inspiration from prior work that combines shape domains and data domains, such as the product domain studied in [11]. Another interesting direction for future research is whether the idea underlying shape extrapolation has application in the synthesis of heap-manipulating programs, e.g., as studied in [21].

Acknowledgments. This work was supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. The work of the Austrian team leading to this result has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 101034440. The collaboration of the teams was also partially supported under the project VASSAL: “Verification and Analysis for Safety and Security of Applications in Life” funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022. ■

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Code Availability The proof-of-concept implementation of our techniques, called Brush in the text, is available as an artifact [28].

References

1. Berdine, J., Calcagno, C., O’Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS. pp. 97–109. LNCS, Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30538-5_9
2. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Biere, A., Parker, D. (eds.) TACAS. pp. 347–367. LNCS, Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_21
3. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Finkbeiner, B., Kovács, L. (eds.) TACAS. pp. 299–329. LNCS, Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57256-2_15
4. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 459–465. LNCS, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
5. Calcagno, C., Distefano, D., O’Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. SIGPLAN Not. **44**(1), 289–300 (2009). <https://doi.org/10.1145/1594834.1480917>
6. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6) (2011). <https://doi.org/10.1145/2049697.2049700>
7. Calcagno, C., O’Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS. pp. 366–378 (2007). <https://doi.org/10.1109/LICS.2007.30>
8. Distefano, D., O’Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS. pp. 287–302. LNCS, Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11691372_19
9. Drăgoi, C., Enea, C., Sighireanu, M.: Local shape analysis for overlaid data structures. In: SAS. pp. 150–171. LNCS, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_10
10. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Logozzo, F., Fähndrich, M. (eds.) SAS. pp. 215–237. LNCS, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_13
11. Giet, J., Ridoux, F., Rival, X.: A product of shape and sequence abstractions. In: Hermenegildo, M.V., Morales, J.F. (eds.) SAS. pp. 310–342. LNCS, Springer, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_15

12. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. In: Palsberg, J., Su, Z. (eds.) SAS. pp. 188–204. LNCS, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_14
13. Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T., Zuleger, F.: Low-Level Bi-Abduction. In: Ali, K., Vitek, J. (eds.) ECOOP. LIPIcs, vol. 222, pp. 19:1–19:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2022). <https://doi.org/10.4230/LIPIcs.ECOOP.2022.19>
14. Illous, H., Lemerre, M., Rival, X.: A relational shape abstract domain. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods, pp. 212–229. LNCS, Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_15
15. Illous, H., Lemerre, M., Rival, X.: Interprocedural shape analysis using separation logic-based transformer summaries. In: Pichardie, D., Sighireanu, M. (eds.) SAS. pp. 248–273. LNCS, Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65474-0_12
16. Kaindlstorfer, D.: Enhancing Abstraction and Symbolic Execution for Shape Analysis of C-Programs operating on Linked Lists. Diploma thesis, TU Wien (2023). <https://doi.org/10.34726/hss.2023.109623>
17. Le, Q.L., Gherghina, C., Qin, S., Chin, W.N.: Shape analysis via second-order bi-abduction. In: Biere, A., Bloem, R. (eds.) CAV. pp. 52–68. LNCS, Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_4
18. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O’Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proc. ACM Program. Lang. **6**(OOPSLA1) (2022). <https://doi.org/10.1145/3527325>
19. Magill, S., Nanevski, A., Clarke, E.M., Lee, P.: Inferring invariants in separation logic for imperative list-processing programs (2015), draft
20. Maksimović, P., Cronjäger, C., Löw, A., Sutherland, J., Gardner, P.: Exact separation logic: Towards bridging the gap between verification and bug-finding. In: Ali, K., Salvaneschi, G. (eds.) ECOOP. LIPIcs, vol. 263, pp. 19:1–19:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2023). <https://doi.org/10.4230/LIPIcs.ECOOP.2023.19>
21. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **3**(POPL) (2019). <https://doi.org/10.1145/3290385>
22. Qin, S., He, G., Chin, W.N., Craciun, F., He, M., Ming, Z.: Automated specification inference in a combined domain via user-defined predicates. Sci. Comput. Program. **148**(C), 189–212 (2017). <https://doi.org/10.1016/j.scico.2017.05.007>
23. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O’Hearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: CAV. pp. 225–252. LNCS, Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53291-8_14
24. Raad, A., Berdine, J., Dreyer, D., O’Hearn, P.W.: Concurrent incorrectness separation logic. Proc. ACM Program. Lang. **6**(POPL) (2022). <https://doi.org/10.1145/3498695>
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society, USA (2002). <https://doi.org/10.1109/LICS.2002.1029817>
26. Rysavy, L.: Join operators for bi-abductive analysis of low-level code. Diploma thesis, TU Wien (2024). <https://doi.org/10.34726/hss.2024.119373>
27. Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., Garg, D.: RefinedC: automating the foundational verification of c code with refined ownership types. In: PLDI. pp. 158–174. Association for Computing Machinery, New York (2021). <https://doi.org/10.1145/3453483.3454036>

28. Sextl, F., Rogalewicz, A., Vojnar, T., Zuleger, F.: Artifact for "compositional shape analysis with shared abduction and biabductive loop acceleration" (2025). <https://doi.org/10.5281/zenodo.14623977>
29. Sextl, F., Rogalewicz, A., Vojnar, T., Zuleger, F.: Compositional shape analysis with shared abduction and biabductive loop acceleration (Extended Version) (2024), <https://arxiv.org/abs/2307.06346>
30. Spies, S., Gäher, L., Sammler, M., Dreyer, D.: Quiver: Guided abductive inference of separation logic specifications in coq. Proc. ACM Program. Lang. **8**(PLDI) (2024). <https://doi.org/10.1145/3656413>
31. Wyatt, P.: Avoiding game crashes related to linked lists (2012), <http://www.codeofhonor.com/blog/avoiding-game-crashes-related-to-linked-lists>
32. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. **7**(OOPSLA1) (2023). <https://doi.org/10.1145/3586045>
33. Zilberstein, N., Saliling, A., Silva, A.: Outcome separation logic: Local reasoning for correctness and incorrectness with computational effects. Proc. ACM Program. Lang. **8**(OOPSLA1) (2024). <https://doi.org/10.1145/3649821>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





SMT-Boosted Security Types for Low-Level MPC

Christian Skalka^(✉) and Joseph P. Near

University of Vermont, Burlington, VT 05405, USA
ceskalka@uvm.edu jnear@uvm.edu

Abstract. Secure Multi-Party Computation (MPC) is an important enabling technology for data privacy in modern distributed applications. We develop a new type theory to automatically enforce correctness, confidentiality, and integrity properties of protocols written in the *Prelude/Overture* language framework. Judgements in the type theory are predicated on SMT verifications in a theory of finite fields, which supports precise and efficient analysis. Our approach is automated, compositional, scalable, and generalizes to arbitrary prime fields for data and key sizes.

1 Introduction

Data privacy is a critical concern in distributed applications including privacy-preserving machine learning [15, 16, 18, 20] and blockchains [12, 14, 21, 30]. Secure multiparty computation (MPC) is a key software-based enabling technology for these applications. MPC protocols provide both confidentiality and integrity properties, formulated as real/ideal (aka simulator) security and universal composability (UC) [11], with well-studied manual proof methods [19]. MPC security semantics has also been reformulated as *hyperproperties* [3, 13, 28] to support automated language-based proof methods. Recent research in the SMT community has developed theories of finite fields [24] with clear relevance to verification of cryptographic schemes that rely on field arithmetic. The goal of this paper is to combine these two approaches in a decidable type system for verifying correctness and security properties of MPC protocols.

Our focus is on low-level MPC protocols. Previous high-level MPC-enabled languages such as Wysteria [25] and Viaduct [3] are designed to provide effective programming of full applications, and incorporate sophisticated compilation techniques such as orchestration [2] to guarantee high-level security properties. But these approaches rely on *libraries* of low-level MPC protocols, such as binary and arithmetic circuits. These low-level protocols are probabilistic, encapsulate abstractions such as secret sharing and semi-homomorphic encryption, and are complementary to high-level language design.

1.1 Overview and Contributions

Our work is based on the *Prelude/Overture* framework developed in previous work [28]. The prior work developed the language and semantics, plus two mech-

anisms for verifying security: an automated approach for protocols in $\mathbb{F}_2$ (the binary field), based on enumerating all possible executions of the protocol, that scales only to very small protocols; and a manual approach based on a program logic. In Section 2 we recall the *Overture* language model and key definitions. In Section 3 we recall formulations of program distributions, and key probabilistic hyperproperties of MPC in *Overture*.

Together, our contributions *automate* the verification of security properties in *Prelude/Overture*. In this work, we build on [28] by developing automatic verification approaches that work for arbitrary finite fields and scale to large protocols. To accomplish this, we develop new type systems for compositional verification, and use SMT to automate the checks that were accomplished by enumeration in prior work. Specifically, we make the following contributions:

- i. In Section 4, an interpretation of protocols in *Overture* as SMT constraints (Theorem 4.2) and methodology for verifying correctness properties.
- ii. In Section 5, a confidentiality type system for *Overture* protocols with a soundness property guaranteeing that information is released only through explicit declassifications (Theorem 5.1).
- iii. In Section 6, an integrity type system for *Overture* protocols with a soundness property guaranteeing robustness in malicious settings (Theorem 6.1).
- iv. In Section 7, a dependent Hoare type system and algorithm for the *Prelude* language that is sound for the confidentiality and integrity type systems (Theorem 7.1), with independently verified, compositional Π types for functions.

In all cases, our type systems are boosted by verification mechanisms provided by Satisfiability Modulo Finite Fields [24]. In Section 8 we develop and discuss evaluation and implementation of the Goldreich-Micali-Wigderson (GMW) protocol. In an extended version of this paper we also discuss the Bendlin-Damgard-Orland-Zakarias (BDOZ) protocol [29].

1.2 Related Work

The goal of the *Overture/Prelude* framework is to automate correctness and security verification of low-level protocols. Other prior work has considered automated verification of high-level protocols and manual verification of low-level protocols.

The most closely related work is [28], where foundations for *Overture/Prelude* are introduced including a verification algorithm. However, this algorithm only works for $\mathbb{F}_2$ —e.g., binary circuits— and does not scale due to exponential complexity, necessitating semi-automated proof techniques. In contrast, our type systems scale to arbitrary prime fields and whole program analysis for larger circuits (protocols). So our work is a significant advancement of *Overture/Prelude*. Our work is also closely related to Satisfiability Modulo Finite Fields [24] but only in the sense that we apply their method as we discuss in Section 4 and subsequently.

Other low-level languages with probabilistic features for privacy preserving protocols have been proposed. The λ_{obliv} language [9] uses a type system to

automatically enforce so-called probabilistic trace obliviousness. But similar to previous work on oblivious data structures [31], obliviousness is related to pure noninterference, not the conditional form related to passive MPC security (Definition 3.8). The Haskell-based security type system in [22] enforces a version of noninterference that is sound for passive security, but does not consider malicious security. And properties of real/ideal passive and malicious security for a probabilistic language have been (manually) formulated in EasyCrypt [13].

Several high-level languages have been developed for writing MPC applications. Previous work on analysis for the SecreC language [4, 6] is concerned with properties of complex MPC circuits, in particular a user-friendly specification and automated enforcement of declassification bounds in programs that use MPC in subprograms. The Wys* language [26], based on Wysteria [25], has similar goals and includes a trace-based semantics for reasoning about the interactions of MPC protocols. Their compiler also guarantees that underlying multi-threaded protocols enforce the single-threaded source language semantics. These two lines of work were focused on passive security. The Viaduct language [3] has a well-developed information flow type system that automatically enforces both confidentiality and integrity through hyperproperties such as robust declassification, in addition to rigorous compilation guarantees through orchestration [2]. However, these high level languages lack probabilistic features and other abstractions of low-level protocols, the implementation and security of which are typically assumed as a selection of library components.

Program logics for probabilistic languages and specifically reasoning about properties such as joint probabilistic independence is also important related work. Probabilistic Separation Logic (PSL) [1] develops a logical framework for reasoning about probabilistic independence (aka separation) in programs, and they consider several (hyper)properties, such as perfect secrecy of one-time-pads and indistinguishability in secret sharing, that are critical to MPC. Lilac [17] extends this line of work with formalisms for conditional independence which is also particularly important for MPC [28].

The Cryptol and SAW tools [10] allow programmers to program security protocols that are verified using SMT. However they are used for cryptographic protocols more generally, and are not designed specifically for MPC as our framework is.

2 *Overture* Syntax and Operational Semantics

The *Overture* language establishes a basic model of synchronous protocols between a federation of *clients* exchanging values in the binary field. We identify clients by natural numbers and federations- finite sets of clients- are always given statically. Our threat model assumes a partition of the federation into *honest* H and *corrupt* C subsets. We model probabilistic programming via a *random tape* semantics. That is, we will assume that programs can make reference to values chosen from a uniform random distributions defined in the initial program memory. Programs aka protocols execute deterministically given the random tape.

2.1 Syntax

$v \in \mathbb{F}_p, w \in \text{String}, \iota \in \text{Clients} \subset \mathbb{N}$	
$\varepsilon ::= v \mid \mathbf{r}[w] \mid \mathbf{s}[w] \mid \mathbf{m}[w] \mid \mathbf{p}[w] \mid \varepsilon - \varepsilon \mid \varepsilon + \varepsilon \mid \varepsilon * \varepsilon \mid \text{OT}(\varepsilon @ \iota, \varepsilon, \varepsilon)$	<i>expressions</i>
$x ::= \mathbf{r}[w] @ \iota \mid \mathbf{s}[w] @ \iota \mid \mathbf{m}[w] @ \iota \mid \mathbf{p}[w] \mid \text{out} @ \iota$	<i>variables</i>
$\pi ::= \mathbf{m}[w] @ \iota := \varepsilon @ \iota \mid \mathbf{p}[w] := \varepsilon @ \iota \mid \text{out} @ \iota := \varepsilon @ \iota \mid \pi; \pi$	<i>protocols</i>

Fig. 1. Syntax of *Overture*

The syntax of *Overture*, defined in Figure 1, includes values v and standard operations of addition, subtraction, and multiplication in a finite field $\mathbb{F}_p$ where p is some prime. Protocols are given input secret values $\mathbf{s}[w]$ as well as random samples $\mathbf{r}[w]$ on the input tape, implemented using a *memory* as described below (Section 2.2) where w is a distinguishing identifier string. Protocols are sequences of assignment commands of three different forms:

- $\mathbf{m}[w] @ \iota_2 := \varepsilon @ \iota_1$: This is a *message send* where expression ε is computed by client ι_1 and sent to client ι_2 as message $\mathbf{m}[w]$.
- $\mathbf{p}[w] := \varepsilon @ \iota$: This is a *public reveal* where expression ε is computed by client ι and broadcast to the federation, typically to communicate intermediate results for use in final output computations.
- $\text{out} @ \iota := \varepsilon @ \iota$: This is an *output* where expression ε is computed by client ι and reported as its output. As a sanity condition we disallow commands $\text{out} @ \iota_1 := \varepsilon @ \iota_2$ where $\iota_1 \neq \iota_2$.

The distinction between messages and broadcast public reveal is consistent with previous formulations, e.g., [22]. To identify and distinguish between collections of variables in protocols we introduce the following notation.

Definition 2.1. We let x range over variables which are identifiers where client ownership is specified- e.g., $\mathbf{m}[\text{foo}] @ \iota$ is a message *foo* that was sent to ι . We let X range over sets of variables, and more specifically, S ranges over sets of secret variables $\mathbf{s}[w] @ \iota$, R ranges over sets of random variables $\mathbf{r}[w] @ \iota$, M ranges over sets of message variables $\mathbf{m}[w] @ \iota$, P ranges over sets of public variables $\mathbf{p}[w]$, and O ranges over sets of output variables $\text{out} @ \iota$. Given a program π , we write $\text{iovars}(\pi)$ to denote the particular set $S \cup M \cup P \cup O$ of variables in π and $\text{secrets}(\pi)$ to denote S , and we write $\text{rands}(\pi)$ to denote the particular set R of random samplings in π . We write $\text{vars}(\pi)$ to denote $\text{iovars}(\pi) \cup \text{rands}(\pi)$. For any set of variables X and clients I , we write X_I to denote the subset of X owned by any client $\iota \in I$, in particular we write X_H and X_C to denote the subsets belonging to honest and corrupt parties, respectively.

In all cases we disallow overwriting of variables, since assignment is intended to represent distinct message sends. An example of a *Overture* protocol, for additive secret sharing, is discussed in Section 4.2.

2.2 Semantics

$$\begin{array}{ll}
\llbracket \sigma, v \rrbracket_\iota = v & \llbracket \sigma, \mathbf{r}[w] \rrbracket_\iota = \sigma(\mathbf{r}[w] \mathbb{Q}_\iota) \\
\llbracket \sigma, \varepsilon_1 + \varepsilon_2 \rrbracket_\iota = \llbracket \llbracket \sigma, \varepsilon_1 \rrbracket_\iota + \llbracket \sigma, \varepsilon_2 \rrbracket_\iota \rrbracket & \llbracket \sigma, \mathbf{s}[w] \rrbracket_\iota = \sigma(\mathbf{s}[w] \mathbb{Q}_\iota) \\
\llbracket \sigma, \varepsilon_1 - \varepsilon_2 \rrbracket_\iota = \llbracket \llbracket \sigma, \varepsilon_1 \rrbracket_\iota - \llbracket \sigma, \varepsilon_2 \rrbracket_\iota \rrbracket & \llbracket \sigma, \mathbf{m}[w] \rrbracket_\iota = \sigma(\mathbf{m}[w] \mathbb{Q}_\iota) \\
\llbracket \sigma, \varepsilon_1 * \varepsilon_2 \rrbracket_\iota = \llbracket \llbracket \sigma, \varepsilon_1 \rrbracket_\iota * \llbracket \sigma, \varepsilon_2 \rrbracket_\iota \rrbracket & \llbracket \sigma, \mathbf{p}[w] \rrbracket_\iota = \sigma(\mathbf{p}[w])
\end{array}$$

$$(\sigma, x := \varepsilon \mathbb{Q}_\iota) \Rightarrow \sigma \{x \mapsto \llbracket \sigma, \varepsilon \rrbracket_\iota\} \quad \frac{(\sigma_1, \pi_1) \Rightarrow \sigma_2 \quad (\sigma_2, \pi_2) \Rightarrow \sigma_3}{(\sigma_1, \pi_1; \pi_2) \Rightarrow \sigma_3}$$

Fig. 2. Semantics of *Overture* expressions (T) and programs (B).

Memories are fundamental to the semantics of *Overture* and provide random tape and secret inputs to protocols, and also record message sends, public broadcast, and client outputs.

Definition 2.2. *Memories* σ are finite (partial) mappings from variables x to values $v \in \mathbb{F}_p$. The domain of a memory is written $\text{dom}(\sigma)$ and is the finite set of variables on which the memory is defined. We write $\sigma \{x \mapsto v\}$ for $x \notin \text{dom}(\sigma)$ to denote the memory σ' such that $\sigma'(x) = v$ and otherwise $\sigma'(y) = \sigma(y)$ for all $y \in \text{dom}(\sigma)$. We write $\sigma \subseteq \sigma'$ iff $\text{dom}(\sigma) \subseteq \text{dom}(\sigma')$ and $\sigma(x) = \sigma'(x)$ for all $x \in \text{dom}(\sigma)$. Given any σ and σ' with $\sigma(x) = \sigma'(x)$ for all $x \in \text{dom}(\sigma) \cap \text{dom}(\sigma')$, we write $\sigma \uplus \sigma'$ to denote the memory with domain $X = \text{dom}(\sigma) \cup \text{dom}(\sigma')$ such that:

$$\forall x \in X. (\sigma \uplus \sigma')(x) = \begin{cases} \sigma(x) & \text{if } x \in \text{dom}(\sigma) \\ \sigma'(x) & \text{otherwise} \end{cases}$$

In our subsequent presentation we will often want to consider arbitrary memories that range over particular variables and to restrict memories to particular subsets of their domain:

Definition 2.3. Given a set of variables X and memory σ , we write σ_X to denote the memory with $\text{dom}(\sigma_X) = X$ and $\sigma_X(x) = \sigma(x)$ for all $x \in X$. We define $\text{mems}(X)$ as the set of all memories with domain X :

$$\text{mems}(X) \triangleq \{\sigma \mid \text{dom}(\sigma) = X\}$$

So for example, the set of all random tapes for a protocol π is $\text{mems}(\text{rands}(\pi))$, and the memory $\sigma_{\text{secrets}(\pi)}$ is σ restricted to the secrets in π .

Given a variable-free expression ε , we write $\llbracket \varepsilon \rrbracket$ to denote the standard interpretation of ε in the arithmetic field $\mathbb{F}_p$. With the introduction of variables to expressions, we need to interpret variables with respect to a specific memory, and all variables used in an expression must belong to a specified client. Thus, we denote interpretation of expressions ε computed on a client ι as $\llbracket \sigma, \varepsilon \rrbracket_\iota$. This interpretation is defined in Figure 2, along with the big-step reduction relation $\Rightarrow$ to evaluate commands. Reduction is a mapping from *configurations* (σ, π) to final stores where all three command forms- message send, broadcast, and output- are implemented as updates to the memory σ .

Oblivious Transfer A passive secure oblivious transfer (OT) protocol based on previous work [1] can be defined in *Overture*, albeit assuming some shared randomness. Alternatively, a simple passive secure OT can be defined with the addition of public key cryptography as a primitive. But given the diversity of approaches to OT, we instead assume that OT is abstract with respect to its implementation, and any use is of the form $\mathbf{m}[w]\mathbb{Q}\iota_1 := \text{OT}(\varepsilon_1\mathbb{Q}\iota_1, \varepsilon_2, \varepsilon_3)\mathbb{Q}\iota_2$ – given a *choice bit* ε_1 provided by a receiver ι_1 , the sender ι_2 sends either ε_2 or ε_3 . Critically, ι_2 learns nothing about ε_1 and ι_1 learns nothing about the unselected value. We return to the formal interpretation of OT in Section 4.

2.3 *Overture* Adversarial Semantics

$$\begin{array}{c}
\frac{\iota \in H}{(\sigma, x := \varepsilon\mathbb{Q}\iota) \Rightarrow_{\mathcal{A}} \sigma \{x \mapsto \llbracket \sigma, \varepsilon \rrbracket_{\iota}\}} \quad \frac{\iota \in C}{(\sigma, x := \varepsilon\mathbb{Q}\iota) \Rightarrow_{\mathcal{A}} \sigma \{x \mapsto \llbracket \text{rewrite}_{\mathcal{A}}(\sigma_C, \varepsilon) \rrbracket_{\iota}\}} \\
\\
\frac{\llbracket \sigma, \varepsilon_1 \rrbracket_{\iota} = \llbracket \sigma, \varepsilon_2 \rrbracket_{\iota} \text{ or } \iota \in C}{(\sigma, \mathbf{assert}(\varepsilon_1 = \varepsilon_2)\mathbb{Q}\iota) \Rightarrow_{\mathcal{A}} \sigma} \quad \frac{\llbracket \sigma, \varepsilon_1 \rrbracket_{\iota} \neq \llbracket \sigma, \varepsilon_2 \rrbracket_{\iota}}{(\sigma, \mathbf{assert}(\varepsilon_1 = \varepsilon_2)\mathbb{Q}\iota) \Rightarrow_{\mathcal{A}} \perp} \\
\\
\frac{(\sigma_1, \pi_1) \Rightarrow_{\mathcal{A}} \sigma_2 \quad (\sigma_2, \pi_2) \Rightarrow_{\mathcal{A}} \sigma_3}{(\sigma_1, \pi_1; \pi_2) \Rightarrow_{\mathcal{A}} \sigma_3} \quad \frac{(\sigma_1, \pi_1) \Rightarrow_{\mathcal{A}} \perp}{(\sigma_1, \pi_1; \pi_2) \Rightarrow_{\mathcal{A}} \perp} \\
\\
\frac{(\sigma_1, \pi_1) \Rightarrow_{\mathcal{A}} \sigma_2 \quad (\sigma_2, \pi_2) \Rightarrow_{\mathcal{A}} \perp}{(\sigma_1, \pi_1; \pi_2) \Rightarrow_{\mathcal{A}} \sigma_2}
\end{array}$$

Fig. 3. Adversarial semantics of *Overture*.

In the malicious model we assume that corrupt clients are in thrall to an adversary $\mathcal{A}$ that does not necessarily follow the rules of the protocol. We model this by positing a $\text{rewrite}_{\mathcal{A}}$ function which is given a corrupt memory σ_C and expression ε , and returns a rewritten expression that can be interpreted to yield a corrupt input. We define the evaluation relation that incorporates the adversary in Figure 3.

A key technical distinction of the malicious setting is that it typically incorporates “abort”. Honest parties implement strategies to detect rule-breaking— aka *cheating*— by using, e.g., message authentication codes with semi-homomorphic properties as in BDOZ/SPDZ [23]. If cheating is detected, the protocol is aborted. To model this, we extend *Overture* with an **assert** command:

$$\pi ::= \dots \mid \mathbf{assert}(\varepsilon = \varepsilon)\mathbb{Q}\iota$$

As we will discuss, this form of asserting equality between expressions is sufficient to capture MAC checking and support integrity in protocols. The big-step adversarial semantics relation $\Rightarrow_{\mathcal{A}}$ is then defined in Figure 3 as a mapping from configurations to a store σ or $\perp$.

Protocol Runs Our semantics require that random tapes contain values for all program values $\mathbf{r}[w]@l$ sampled from a uniform distribution over $\mathbb{F}_p$. Input memories also contain input secret values and possibly other initial view elements as a result of pre-processing, e.g., Beaver triples for efficient multiplication, and/or MACed share distributions as in BDOZ/SPDZ [11, 23]. We define $runs(\pi)$ as the set of final memories resulting from execution of π given any initial memory, possibly augmented with a preprocessed memory (a memory that satisfies a predicate *preproc*), and $runs_{\mathcal{A}}(\pi)$ is the analog in the malicious setting. Since we constrain programs to not overwrite any variable, we are assured that final memories contain both a complete record of all initial secrets as well as views resulting from communicated information. This formulation will be fundamental to our consideration of probability distributions and hyperproperties of protocols.

Definition 2.4. *Given π with $secrets(\pi) = S$ and $rands(\pi) = R$ and predicate *preproc* on memories, define:*

$$runs(\pi) \triangleq \{\sigma \mid \exists \sigma_1 \in mems(S \cup R). \exists \sigma_2. preproc(\sigma_2) \wedge (\sigma_1 \uplus \sigma_2, \pi) \Rightarrow \sigma\}$$

And similarly for any $\mathcal{A}$ define:

$$runs_{\mathcal{A}}(\pi) \triangleq \{\sigma \mid \exists \sigma_1 \in mems(S \cup R). \exists \sigma_2. preproc(\sigma_2) \wedge (\sigma_1 \uplus \sigma_2, \pi) \Rightarrow_{\mathcal{A}} \sigma\}$$

3 Security Model

MPC protocols are intended to implement some *ideal functionality* $\mathcal{F}$ with per-client outputs. In the *Overture* setting, given a protocol π that implements $\mathcal{F}$, with $iovars(\pi) = S \cup M \cup P \cup O$, the domain of $\mathcal{F}$ is $mems(S)$ and its range is $mems(O)$. Security in the MPC setting means that, given $\sigma \in mems(S)$, a secure protocol π does not reveal any more information about honest secrets σ_H to parties in C beyond what is implicitly declassified by $\mathcal{F}(\sigma)$. Security comes in *passive* and *malicious* flavors, wherein the adversary either follows the rules or not, respectively. Characterization of both real world protocol execution and simulation is defined probabilistically. In this work we will focus on the enforcement of hyperproperties of passive and malicious security developed in related work [28], that were shown to be sound for real/ideal security properties that are traditionally used for MPC. As in that work we use probability mass functions to express joint dependencies between input and output variables, as a metric of information leakage.

3.1 Probability Mass Functions (pmfs)

Pmfs map *realizations* of variables in a joint distribution to values in $[0..1]$. For convenience and following [1, 28] we use memories to represent realizations, so for example given a pmf $\mathcal{P}$ over variables $\{\mathbf{s}[x]@1, \mathbf{m}[y]@2\}$ we write $\mathcal{P}(\{\mathbf{s}[x]@1 \mapsto 0, \mathbf{m}[y]@2 \mapsto 1\})$ to denote the (joint) probability that $\mathbf{s}[x]@1 = 0 \wedge \mathbf{m}[y]@2 = 1$. Recall from Section 2.2 that $\uplus$ denotes the combination of memories, so for example $\{\mathbf{s}[x]@1 \mapsto 0\} \uplus \{\mathbf{m}[y]@2 \mapsto 1\} = \{\mathbf{s}[x]@1 \mapsto 0, \mathbf{m}[y]@2 \mapsto 1\}$.

Definition 3.1. A probability mass function $\mathcal{P}$ is a function mapping memories in $\text{mems}(X)$ for given variables X to values in $\mathbb{R}$ such that:

$$\sum_{\sigma \in \text{mems}(X)} \mathcal{P}(\sigma) = 1$$

Now, we can define a notion of marginal and conditional distributions as follows, which are standard for discrete probability mass functions.

Definition 3.2. Given $\mathcal{P}$ with $\text{dom}(\mathcal{P}) = \text{mems}(X_2)$, the marginal distribution of variables $X_1 \subseteq X_2$ in $\mathcal{P}$ is denoted $\mathcal{P}_{X_1}$ and defined as follows:

$$\forall \sigma \in \text{mems}(X_1) \quad \mathcal{P}_{X_1}(\sigma) \triangleq \sum_{\sigma' \in \text{mems}(X_2 - X_1)} \mathcal{P}(\sigma \uplus \sigma')$$

Definition 3.3. Given $\mathcal{P}$, the conditional distribution of X_1 given X_2 where $X_1 \cup X_2 \subseteq \text{dom}(\mathcal{P})$ and $X_1 \cap X_2 = \emptyset$ is denoted $\mathcal{P}_{(X_1|X_2)}$ and defined as follows:

$$\forall \sigma \in \text{mems}(X_1 \cup X_2) \quad \mathcal{P}_{(X_1|X_2)}(\sigma) \triangleq \begin{cases} 0 & \text{if } \mathcal{P}_{X_2}(\sigma_{X_2}) = 0 \\ \mathcal{P}_{X_1 \cup X_2}(\sigma) / \mathcal{P}_{X_2}(\sigma_{X_2}) & \text{o.w.} \end{cases}$$

We also define some convenient syntactic sugarings. The first will allow us to compare marginal distributions under different realization conditions (as in, e.g., Definition 3.8), the others are standard.

Definition 3.4. Given $\mathcal{P}$, for all $\sigma_1 \in \text{mems}(X_1)$ and $\sigma_2 \in \text{mems}(X_2)$ define:

1. $\mathcal{P}_{(X_1|\sigma_2)}(\sigma_1) \triangleq \mathcal{P}_{(X_1|X_2)}(\sigma_1 \uplus \sigma_2)$
2. $\mathcal{P}(\sigma_1) \triangleq \mathcal{P}_{X_1}(\sigma_1)$
3. $\mathcal{P}(\sigma_1|\sigma_2) \triangleq \mathcal{P}_{(X_1|X_2)}(\sigma_1 \uplus \sigma_2)$

As in previous work [1, 9, 17, 28], probabilistic independence, aka separation, is an important concept and we adopt the following standard notation to express conditional and unconditional separation:

Definition 3.5. We write $\mathcal{P}|X_1 \vdash X_2 * X_3$ iff for all $\sigma \in \text{mems}(X_1 \cup X_2)$ and $\sigma' \in \text{mems}(X_3)$ we have $\mathcal{P}(\sigma|\sigma') = \mathcal{P}(\sigma_{X_1}|\sigma') * \mathcal{P}(\sigma_{X_2}|\sigma')$. If X_1 is empty we write $\mathcal{P} \vdash X_2 * X_3$.

3.2 Basic Distribution of a Protocol

We treat all elements of $\text{runs}(\pi)$ as equally likely. This establishes the basic program distribution that can be marginalized and conditioned to quantify input/output information dependencies.

Definition 3.6. The basic distribution of π , written $\text{BD}(\pi)$, is defined such that for all $\sigma \in \text{mems}(\text{vars}(\pi))$:

$$\text{BD}(\pi)(\sigma) = 1/|\text{runs}(\pi)| \text{ if } \sigma \in \text{runs}(\pi), \text{ otherwise } 0$$

For any $\mathcal{A}$ the basic $\mathcal{A}$ distribution of π , written $\text{BD}(\pi, \mathcal{A})$, is defined such that for all $\sigma \in \text{mems}(\text{iovars}(\pi) \cup R)$:

$$\text{BD}(\pi, \mathcal{A})(\sigma) = 1/|\text{runs}_{\mathcal{A}, \perp}(\pi)| \text{ if } \sigma \in \text{runs}_{\mathcal{A}, \perp}(\pi), \text{ otherwise } 0$$

where $\text{runs}_{\mathcal{A}, \perp}$ pads out undefined views and outputs with $\perp$:

$$\begin{aligned} \text{runs}_{\mathcal{A}, \perp}(\pi) &\triangleq \\ \{\sigma \{x_1 \mapsto \perp, \dots, x_n \mapsto \perp\} \mid \sigma \in \text{runs}_{\mathcal{A}}(\pi) \wedge \{x_1, \dots, x_n\} = (V \cup O) - \text{dom}(\sigma)\} \end{aligned}$$

3.3 Honest and Corrupt Views

Information about honest secrets can be revealed to corrupt clients through messages sent from honest to corrupt clients, and through publicly broadcast information from honest clients. Dually, corrupt clients can impact protocol integrity through the messages sent from corrupt to honest clients, and through publicly broadcast information from corrupt clients. We call the former *corrupt views*, and the latter *honest views*. We let V range over sets of views, i.e., subsets of $M \cup P$.

Definition 3.7 (Corrupt and Honest Views). We let V range over views which are sets of messages and reveals. Given a program π with $\text{iovars}(\pi) = S \cup M \cup P \cup O$, define $\text{views}(\pi) \triangleq M \cup P$, and define $V_{H \triangleright C}$ as the messages and reveals in $V = M \cup P$ sent from honest to corrupt parties, called corrupt views:

$$\begin{aligned} V_{H \triangleright C} &\triangleq \{ p[w] \mid p[w] := \varepsilon @ \iota \in \pi \wedge \iota \in H \} \cup \\ &\quad \{ m[w] @ \iota \mid m[w] @ \iota := \varepsilon @ \iota' \in \pi \wedge \iota \in C \wedge \iota' \in H \} \end{aligned}$$

and similarly define $V_{C \triangleright H}$ as the subset of V sent from corrupt to honest parties, called honest views:

$$\begin{aligned} V_{C \triangleright H} &\triangleq \{ p[w] \mid p[w] := \varepsilon @ \iota \in \pi \wedge \iota \in C \} \cup \\ &\quad \{ m[w] @ \iota \mid m[w] @ \iota := \varepsilon @ \iota' \in \pi \wedge \iota \in H \wedge \iota' \in C \} \end{aligned}$$

3.4 Hyperproperties of Confidentiality and Integrity

In this section we restate hyperproperties developed in previous work [28]. The reader is referred to that paper for a more thorough discussion of their relation to real/ideal security. In this paper, we leverage that relation to establish security properties in our type system.

Confidentiality Since MPC protocols release some information about secrets through outputs of $\mathcal{F}$, they do not enjoy strict noninterference. Public reveals and protocol outputs are fundamentally forms of declassification. But the following property of probabilistic noninterference *conditioned on output* is sound for passive security [28].

Definition 3.8 (Noninterference modulo output). *We say that a program π with $\text{iovars}(\pi) = S \cup V \cup O$ satisfies noninterference modulo output iff for all H and C and $\sigma_1 \in \text{mems}(S_C \cup O)$ and $\sigma_2 \in \text{mems}(V_{H \triangleright C})$ we have:*

$$\text{BD}(\pi)_{(S_H | \sigma_1)} = \text{BD}(\pi)_{(S_H | \sigma_1 \uplus \sigma_2)}$$

This conditional noninterference property implies that corrupt views give the adversary no better chance of guessing honest secrets than just the output and corrupt inputs do.

In practice, MPC protocols typically satisfy a *gradual release* property [27], where messages exchanged remain probabilistically separable from secrets, with only declassification events (reveals and outputs) releasing information about honest secrets.

Definition 3.9. *Given H, C , a protocol π with $\text{iovars}(\pi) = S \cup M \cup P \cup O$ satisfies gradual release iff $\text{BD}(\pi) \vdash M_C * S_H$.*

Integrity *Integrity* is an important hyperproperty in security models that admit malicious adversaries. Consistent with formulations in deterministic settings, we define protocol integrity as the preservation of high equivalence (of secrets and views). Intuitively, this property says that any adversarial strategy either “mimics” a passive strategy with some choice of inputs or causes an abort.

Definition 3.10 (Integrity). *We say that a protocol π with $\text{iovars}(\pi) = S \cup V \cup O$ has integrity iff for all H, C , and $\mathcal{A}$, if $\sigma \in \text{runs}_{\mathcal{A}}(\pi)$ then there exists $\sigma' \in \text{mems}(S)$ with $\sigma_{S_H} = \sigma'_{S_H}$ and:*

$$\text{BD}(\pi, \mathcal{A})_{(X | \sigma_{S_H \cup V_{C \triangleright H}})} = \text{BD}(\pi)_{(X | \sigma')}$$

where $X \triangleq (V_{H \triangleright C} \cup O_H) \cap \text{dom}(\sigma)$.

Integrity plus noninterference modulo output implies malicious security in the real/ideal model [28], and integrity plus gradual release is a probabilistic form of robust declassification [27].

4 Overture Constraint Verification

In previous work it was shown that the semantics of *Overture* in $\mathbb{F}_2$ can be implemented with Datalog [28], which is a form of constraint programming. In this paper we extend this idea to arbitrary prime fields by using a more general form of SMT constraint programming, Satisfiability Modulo Finite Fields [24]. As we will show in subsequent sections, this interpretation supports correctness guarantees, and also static type analyses for enforcing confidentiality and integrity properties.

4.1 Constraint Satisfiability Modulo Finite Fields

We introduce the following syntax of SMT-style constraints for *Overture*:

$$\phi ::= x \mid \phi + \phi \mid \phi - \phi \mid \phi * \phi \quad E ::= \phi \equiv \phi \mid E \wedge E \mid \neg E$$

Note that constraint terms ϕ are similar to expressions ε except that ϕ can “cross party lines”. This is needed to express correctness properties—for example, in the Goldreich-Micali-Wigderson (GMW) protocol wire values in circuits are represented by reconstructive shares [11]. If by convention shares of values n are represented by $\mathbf{m}[n]$ on clients, then assuming two clients $\{1, 2\}$ the reconstructed value can be expressed as $\mathbf{m}[n] \mathbf{0}1 + \mathbf{m}[n] \mathbf{0}2$. So, while summing values across clients is disallowed in *Overture* protocols, we can express properties of shared values via constraints.

This language of constraints has an obvious direct interpretation in Satisfiability Modulo Finite Fields [24]. We can leverage this to implement a critical *entailment* property, written $E_1 \models E_2$. Our entailment relation is based on satisfiability in a standard sense, where we represent models as memories σ (mappings from variables to values).

Definition 4.1. We write $\sigma \models E$ iff σ satisfies, aka is a model of, E . We write $E_1 \models E_2$ iff $\sigma \models E_1$ implies $\sigma \models E_2$ for all σ , and note that this relation is a preorder.

Given this Definition, the following Theorem is well-known and a fundamental technique in SMT to implement our (common) notion of entailment.

Theorem 4.1. $E_1 \models E_2$ iff $E_1 \wedge \neg E_2$ is not satisfiable.

4.2 Programs as Constraint Systems

$$\begin{aligned} \lfloor x \rfloor &= x & \lfloor \varepsilon_1 + \varepsilon_2 \mathbf{0}l \rfloor &= (\lfloor \varepsilon_1 \mathbf{0}l \rfloor + \lfloor \varepsilon_2 \mathbf{0}l \rfloor) & \lfloor \varepsilon_1 - \varepsilon_2 \mathbf{0}l \rfloor &= (\lfloor \varepsilon_1 \mathbf{0}l \rfloor - \lfloor \varepsilon_2 \mathbf{0}l \rfloor) \\ \lfloor \varepsilon_1 * \varepsilon_2 \mathbf{0}l \rfloor &= (\lfloor \varepsilon_1 \mathbf{0}l \rfloor * \lfloor \varepsilon_2 \mathbf{0}l \rfloor) \\ \lfloor x := \varepsilon \mathbf{0}l \rfloor &= (x \equiv \lfloor \varepsilon \mathbf{0}l \rfloor) & \lfloor \mathbf{assert}(\varepsilon_1 = \varepsilon_2) \mathbf{0}l \rfloor &= (\lfloor \varepsilon_1 \mathbf{0}l \rfloor \equiv \lfloor \varepsilon_2 \mathbf{0}l \rfloor) \\ \lfloor \pi_1; \pi_2 \rfloor &= (\lfloor \pi_1 \rfloor \wedge \lfloor \pi_2 \rfloor) \end{aligned}$$

Fig. 4. Interpretation of *Overture* expressions (T) and programs (B) as constraints

A central idea of our approach is that we can interpret any protocol π as a set of equality constraints (denoted $\lfloor \pi \rfloor$) and use an SMT solver to verify properties relevant to correctness, confidentiality, and integrity. Further, we can leverage entailment for tractability—we can use annotations to obtain a weakened precondition for relevant properties. That is, given π , program annotations or

other cues can be used to find a minimal E with $\lfloor \pi \rfloor \models E$ for verifying correctness and security.

The mapping $\lfloor \cdot \rfloor$ from programs π to constraints is defined in Figure 4. The interpretation of OT is omitted from this figure which is general. For $\mathbb{F}_2$ (where we allow negation $\neg$ of expressions) the interpretation is:

$$\lfloor \text{OT}(\varepsilon_1 @ \iota_1, \varepsilon_2, \varepsilon_3) @ \iota_2 \rfloor = (\lfloor \varepsilon_1 @ \iota_1 \rfloor * \lfloor \varepsilon_3 @ \iota_2 \rfloor) + (\neg \lfloor \varepsilon_1 @ \iota_1 \rfloor * \lfloor \varepsilon_2 @ \iota_2 \rfloor)$$

In general we will assume that any preprocessing predicate is defined via a constraint E_{pre} , i.e., $preproc(\sigma) \iff \sigma \models E_{pre}$ for all σ . The correctness of the SMT interpretation of programs is characterized as follows.

Theorem 4.2. σ is a model of $E_{pre} \wedge \lfloor \pi \rfloor$ iff $\sigma \in runs(\pi)$.

Example: Correctness of 3-Party Addition Consider the following 3-party additive secret sharing protocol, where each party generates three additive, information theoretically secure secret shares by subtracting random field values used as one-time-pads. Two shares are communicated to the other clients and one remains local. This technique is information-theoretically secure [11] due to the one-time use of random values— that is, < 3 shares reveal nothing about the secret, but the sum of all three *is* the secret. Each party publicly reveals the sums all their shares and then outputs the sum of the reveals.

$$\begin{array}{ll} m[s1]@2 := (s[1] - r[local] - r[x])@1 & p[1] := (r[local] + m[s2] + m[s3])@1 \\ m[s1]@3 := r[x]@1 & p[2] := (m[s1] + r[local] + m[s3])@2 \\ m[s2]@1 := (s[2] - r[local] - r[x])@2 & p[3] := (m[s1] + m[s2] + r[local])@3 \\ m[s2]@3 := r[x]@2 & out@1 := (p[1] + p[2] + p[3])@1 \\ m[s3]@1 := (s[3] - r[local] - r[x])@3 & out@2 := (p[1] + p[2] + p[3])@2 \\ m[s3]@2 := r[x]@3 & out@3 := (p[1] + p[2] + p[3])@3 \end{array}$$

Letting π be this protocol, we can verify correctness as follows— that is, the output sum of all shares is the sum of the three input secrets:

$$\lfloor \pi \rfloor \models out@1 \equiv s[1]@1 + s[2]@2 + s[3]@3$$

5 Confidentiality Types

It is well-known that adding or subtracting a sample from a uniform distribution in a finite field yields a value in a uniform distribution, meaning that samples can be used as one-time-pads with perfect secrecy [1, 9]. In our type system for confidentiality, we aim to approximate distributions by tracking which program variables an expression may depend on. But we also want to capture this fundamental mechanism of encryption. As we will show, SMT can be used in this and other contexts to support confidentiality properties of protocols.

For example, given some string y and a message send such as:

$$m[w]@1 := s[w] * r[w]@2$$

we would assign the following type to denote that $\mathbf{m}[w]\textcircled{1}$ is dependent on both $\mathbf{s}[w]\textcircled{2}$ and $\mathbf{r}[w]\textcircled{2}$:

$$\mathbf{m}[w]\textcircled{1} : \{\mathbf{s}[w]\textcircled{2}, \mathbf{r}[w]\textcircled{2}\}$$

But in the case of 2-party reconstructive sharing:

$$\mathbf{m}[w]\textcircled{1} := \mathbf{s}[w] - \mathbf{r}[w]\textcircled{2}$$

we want the type of $\mathbf{m}[w]\textcircled{1}$ to be an “encrypted” type that is by itself independent of $\mathbf{s}[w]\textcircled{2}$. However, we also want to track the *joint* dependence of $\mathbf{m}[w]\textcircled{2}$ on $\mathbf{r}[w]\textcircled{2}$ and $\mathbf{s}[w]\textcircled{2}$, in case information about $\mathbf{r}[w]\textcircled{2}$ is subsequently leaked. In short, we assign $\mathbf{m}[w]\textcircled{1}$ a “ciphertext type”, aka a *ciphertext*:

$$\mathbf{m}[w]\textcircled{1} : c(\mathbf{r}[w]\textcircled{2}, \{\mathbf{s}[w]\textcircled{2}\})$$

indicating syntactically that $\mathbf{m}[w]\textcircled{1}$ is an encryption of a value dependent on $\mathbf{s}[w]\textcircled{2}$ with $\mathbf{r}[w]\textcircled{2}$.

Of course, there are other methods for encrypting values in MPC protocols—but we can observe that many are algebraically equivalent to this fundamental schema. For example, in Yao’s Garbled Circuits (YGC) in $\mathbb{F}_2$, the “garbler” represents encrypted wire values as a random sample (for 1) or its negation (for 0), and shares its secret input encoding with the “evaluator” by using it as a selection bit. That is, assuming that client 2 is the garbler and client 1 is the evaluator, we can define:

$$\mathbf{m}[w]\textcircled{1} := \text{mux}(\mathbf{s}[w], \neg\mathbf{r}[w], \mathbf{r}[w])\textcircled{2}$$

where for all $\varepsilon_1, \varepsilon_2, \varepsilon_3$ (with $\neg$ denoting negation in $\mathbb{F}_2$):

$$\text{mux}(\varepsilon_1, \varepsilon_2, \varepsilon_3) \triangleq (\neg\varepsilon_1 * \varepsilon_2) + (\varepsilon_1 * \varepsilon_3)$$

and letting this protocol be π the following is valid:

$$[\pi] \models \mathbf{m}[w]\textcircled{1} \equiv \neg\mathbf{s}[w]\textcircled{2} + \mathbf{r}[w]\textcircled{2}$$

This allows us to assign the same ciphertext type to $\mathbf{m}[w]\textcircled{1}$ as above— that is, $c(\mathbf{r}[w]\textcircled{2}, \{\mathbf{s}[w]\textcircled{2}\})$. Since this is a one-time-pad scheme, it is also important that we ensure that samples are used at most once for encryption. As in previous work [9] we use type linearity to enforce this.

The syntax of types T and type environments Γ are presented in Figure 5¹. We also define type judgements for expressions and programs by direct interpretation as terms ϕ and constraints E respectively, via the encoding $[\cdot]$. The ENCODE rule enforces linearity by requiring that variables used for encryption are added to judgements, and guaranteed to not be used elsewhere via a form of union that ensures disjointness of combined sets:

¹ In general we use 2^X to denote the powerset of terms in any grammatical sort χ .

$$\begin{array}{l}
 t ::= x \mid c(x, T) \\
 T \in 2^t \quad \text{confidentiality types} \\
 \Gamma ::= \emptyset \mid \Gamma; x : T \quad \text{confidentiality type environments} \\
 \\
 \text{DEPTY} \quad \frac{}{\emptyset, E \vdash \phi : \text{vars}(\phi)} \quad \text{ENCODE} \quad \frac{E \models \phi \equiv \phi' \oplus \mathbf{r}[w]@u \quad \oplus \in \{+, -\} \quad R, E \vdash \phi' : T}{R; \{\mathbf{r}[w]@u\}, E \vdash \phi : \{c(\mathbf{r}[w]@u, T)\}} \\
 \\
 \text{SEND} \quad \frac{R, E \vdash \phi : T}{R, E \vdash x \equiv \phi : (x : T)} \quad \text{SEQ} \quad \frac{R_1, E \vdash \phi_1 : \Gamma_1 \quad R_2, E \vdash \phi_2 : \Gamma_2}{R_1; R_2, E \vdash \phi_1 \wedge \phi_2 : \Gamma_1; \Gamma_2}
 \end{array}$$

Fig. 5. Syntax and Derivation Rules for *Overture* Confidentiality Types

Definition 5.1. $R_1; R_2 = R_1 \cup R_2$ if $R_1 \cap R_2 = \emptyset$ and is otherwise undefined.

The DEPTY rule handles all cases and just records the dependencies on variables in the term, a conservative approximation of dependencies. The SEND rule records the types of messages, reveals, and outputs in environments, and the SEQ rule combines the types of sequenced programs. Given this we define validity of program type judgements as follows.

Definition 5.2. Given preprocessing predicate E_{pre} and protocol π we say $R, E \vdash E_{pre} \wedge [\pi] : \Gamma$ is valid iff it is derivable and $E_{pre} \wedge [\pi] \models E$.

It is important to note that this presentation of derivations is logical, not algorithmic— in particular it is not syntax-directed due to the ENCODE rule.

5.1 Detecting Leakage Through Dependencies

Given a valid typing, we can leverage structural type information to conservatively approximate the information accessible by the adversary who has control over all corrupt clients in C given a partitioning H, C . In particular, if the adversary has access to an encrypted value and another value that is dependent on its one-time-pad key— i.e., access to values with types of the forms:

$$\{c(\mathbf{r}[w]@u, T)\} \quad \{\mathbf{r}[w]@u, \dots\}$$

then we can conservatively assume that the adversary can decrypt the value of type T and thus have access to those values.

In Figure 6 we define derivation rules for the “leakage” judgement $\Gamma, C \vdash_{leak} T$ which intuitively says that given a protocol with type environment Γ and corrupt clients C , the adversary may be able to observe dependencies on values of type T in their adversarial messages. The real aim of this relation is to determine if $\Gamma, C \vdash_{leak} \{\mathbf{s}[w]@u\}$ with $u \in H^-$ — that is, if the adversary may observe dependencies on honest secrets.

Our main type soundness result combines typing and leakage judgements, and guarantees gradual release of protocols— that is, that adversarial messages reflect

no dependencies on honest secrets. Proof details are provided in an extended version of this paper [29].

Theorem 5.1. *Given π with $\text{views}(\pi) = M \cup P$, if $R, E \vdash E_{pre} \wedge \lfloor \pi \rfloor : \Gamma$ is valid and for all H, C we have $\Gamma, M_C \vdash_{leak} T$ with T closed and $S_H \cap T = \emptyset$, then π satisfies gradual release.*

$$\begin{array}{c}
 \frac{\Gamma \vdash T_1 \rightsquigarrow T_2 \quad \Gamma \vdash T_2 \rightsquigarrow T_3}{\Gamma \vdash T_1 \rightsquigarrow T_3} \qquad \Gamma \vdash T \cup \{\mathbf{m}[w]\mathbf{e}_\ell\} \rightsquigarrow T \cup \Gamma(\mathbf{m}[w]\mathbf{e}_\ell) \\
 \\
 \Gamma \vdash T_1 \cup \{x, c(x, T_2)\} \rightsquigarrow T_1 \cup T_2 \\
 \\
 \Gamma, \emptyset \vdash_{leak} \emptyset \qquad \frac{\Gamma, M \vdash_{leak} T' \quad \Gamma \vdash T' \cup \{x\} \rightsquigarrow T}{\Gamma, M \cup \{x\} \vdash_{leak} T}
 \end{array}$$

Fig. 6. Dependencies in Views: Derivation Rules

It is important to recall from [28] that gradual release does *not* imply passive security, since it only concerns adversarial messages, but not public reveals of outputs where dependencies on input secrets can occur even in secure protocols. Nevertheless, gradual release is an important confidentiality property for MPC (and other distributed) protocols.

6 Integrity Types

Our integrity type system is essentially a taint analysis, with machinery to allow consideration of any H, C partitioning. We also include rules to consider MAC codes to prevent cheating. We consider a semi-homomorphic encryption scheme that is used in BDOZ and SPDZ [5, 7, 8, 23] and is applicable to various species of circuits. However other MAC schemes could be considered with a modified derivation rule incorporated with our taint analysis framework. BDOZ/SPDZ assumes a pre-processing phase that distributes shares with MAC codes to clients. This integrates well with pre-processed Beaver Triples to implement malicious secure, and relatively efficient, multiplication [11].

In BDOZ, a field value v is secret shared among 2 clients with accompanying MAC values. Each client ι gets a pair of values (v_ι, m_ι) , where v_ι is a share of v reconstructed by addition, i.e., $v = \llbracket v_1 + v_2 \rrbracket$, and m_ι is a MAC of v_ι . More precisely, $m_\iota = k + (k_\Delta * v_\iota)$ where *keys* k and k_Δ are known only to $\iota' \neq \iota$ and k_Δ . The *local key* k is unique per MAC, while the *global key* k_Δ is common to all MACs authenticated by ι' . This is a semi-homomorphic encryption scheme that supports addition of shares and multiplication of shares by a constant— for more details the reader is referred to Orsini [23]. We note that while we restrict

$\varsigma ::= \text{High} \mid \text{Low}$		<i>integrity labels</i>	
$\Delta ::= \emptyset \mid \Delta; x : \iota \cdot V$		<i>integrity type environments</i>	
VALUE	SECRET	RANDO	MESG
$\vdash_{\iota} v : \emptyset$	$\vdash_{\iota} \mathbf{s}[w] : \emptyset$	$\vdash_{\iota} \mathbf{r}[w] : \emptyset$	$\vdash_{\iota} \mathbf{m}[w] : \{\mathbf{m}[w]\mathbf{@}\iota\}$
PUBM	BINOP		
$\vdash_{\iota} \mathbf{p}[w] : \{\mathbf{p}[w]\}$	$\frac{\vdash_{\iota} \varepsilon_1 : V_1 \quad \vdash_{\iota} \varepsilon_2 : V_2 \quad \oplus \in \{+, -, *\}}{\vdash_{\iota} \varepsilon_1 \oplus \varepsilon_2 : V_1 \cup V_2}$		
SEND	SEQ		
$\frac{\vdash_{\iota} \varepsilon : V}{E \vdash x := \varepsilon\mathbf{@}\iota : (x : \iota \cdot V)}$	$\frac{E \vdash \pi_1 : \Delta_1 \quad E \vdash \pi_2 : \Delta_2}{E \vdash \pi_1; \pi_2 : \Delta_1; \Delta_2}$		
MAC			
$\frac{E \models [\mathbf{assert}(\psi_{BDOZ}(w))\mathbf{@}\iota]}{E \vdash \mathbf{assert}(\psi_{BDOZ}(w))\mathbf{@}\iota : (\mathbf{m}[ws]\mathbf{@}\iota : \iota \cdot \emptyset)}$			

Fig. 7. Syntax and derivation rule of *Overture* integrity types

values v , m , and k to the same field in this presentation for simplicity, in general m and k can be in extensions of $\mathbb{F}_p$.

As in [28], we use $\langle m[w] \rangle$ to refer to secret-shared values reconstructed with addition, and by convention each share of $\langle m[w] \rangle$ is represented as $m[ws]@_{\iota}$, the MAC of which is represented as a $m[w\mathbf{m}]@_{\iota}$ for all ι , and each client holds a key $m[wk]@_{\iota}$ for authentication of the other's share. Each client also holds a global key $m[\mathbf{delta}]$. For any such share identified by string w , the BDOZ MAC scheme is defined by the equality predicate $\psi_{BDOZ}(w)$:

$$\psi_{BDOZ}(w) \triangleq m[w\mathbf{m}] = m[wk] + (m[\mathbf{delta}] * m[ws])$$

6.1 Typing and Integrity Labeling

The integrity of values during protocol executions depends on the partitioning H, C , since any value received from a member of C is initially considered to be low integrity. But as for confidentiality types, we want our basic type analysis to be generalizable to arbitrary H, C . So integrity type judgements, defined in Figure 7, are of the form $E \vdash \pi : \Delta$, where E plays a similar role as in confidentiality types, as a possibly weakened expression of algebraic properties of π , and environments Δ record dependency information that can be specialized to a given H, C . Integrity environments are ordered lists since order of evaluation is important for tracking integrity—validating low integrity information through MAC checking allows subsequent treatment of it as high integrity, but not before. Note that MAC checking is *not* a form of endorsement as it is usually defined (as a dual of declassification [27]), since a successful check guarantees no deviation from the protocol.

The typing of protocols depends on a type judgement for expressions of the form $\vdash_{\iota} \varepsilon : V$, which records messages and reveals V used in the construction of ε . This source is recorded in Δ for message sends—so for example, for some string w letting:

$$\pi \triangleq (\mathbf{m}[w]@1 := \mathbf{s}[w] - \mathbf{r}[w]@2; \mathbf{p}[w] := \mathbf{m}[w]@1)$$

the following judgement is valid:

$$[\pi] \vdash \pi : (\mathbf{m}[w]@1 : 2 \cdot \emptyset; \mathbf{p}[w] : 1 \cdot \mathbf{m}[w]@1)$$

More formally, we define validity as follows:

Definition 6.1. *Given pre-processing predicate E_{pre} and protocol π , we say $E \vdash \pi : \Delta$ is valid iff it is derivable and $E_{pre} \wedge [\pi] \models E$.*

In the above example, note that the environment implicitly records that $\mathbf{p}[w]$ depends on data from client 1, which in turn depends on data from client 2. The rule for MAC checking allows an overwrite of the dependency. We implement a malicious-secure arithmetic circuit that applies this rule in extended version of this paper [29].

$$\begin{array}{c} \emptyset \rightsquigarrow_{H,C} \mathcal{L}_{H,C} \qquad \frac{\Delta \rightsquigarrow_{H,C} \mathcal{L} \quad \iota \in H}{\Delta; x : \iota \cdot V \rightsquigarrow_{H,C} \mathcal{L}\{x \mapsto \text{High} \wedge (\bigwedge_{x \in V} \mathcal{L}_2(x))\}} \\[1.5cm] \frac{\Delta \rightsquigarrow_{H,C} \mathcal{L} \quad \iota \in C}{\Delta; x : \iota \cdot V \rightsquigarrow_{H,C} \mathcal{L}\{x \mapsto \text{Low}\}} \end{array}$$

Fig. 8. Assigning integrity labels to variables

6.2 Assigning Integrity Labels

In all cases, we want to determine the integrity level of computed values based on assumed partitionings H, C . In the previous example, we want to say that $\mathbf{p}[w]$ has low integrity if either client 1 or 2 is corrupt. To this end we define *security labelings* $\mathcal{L}$ which are mappings from variables x to security levels ς . By default, given a partitioning H, C this labeling maps messages to the integrity level of the owner.

Definition 6.2. *Given H, C , define $\mathcal{L}_{H,C}$ such that $\mathcal{L}_{H,C}(\mathbf{m}[w]@_{\iota}) = \text{High}$ if $\iota \in H$ and Low otherwise for all $\mathbf{m}[w]@_{\iota}$.*

But the values of messages and reveals may be affected by values sent by other parties, which is recorded in the Δ obtained from its typing. To determine whether this impacts integrity for any given H, C , we apply the inductively defined rewrite relation $\Delta \xrightarrow{H, C} \mathcal{L}$, axiomatized in Figure 8. The composition of typing and this latter rewriting obtains essentially a traditional taint analysis, modulo MAC checking. Our main type soundness result for integrity is then given as follows. Proof details are given in [29].

Theorem 6.1. *Given E_{pre} and π with $iovars(\pi) = S \cup V \cup O$, if $E \vdash \pi : \Delta$ is valid and for all H, C with $\Delta \xrightarrow{H, C} \mathcal{L}$ we have $\mathcal{L}(x) = \text{High}$ for all $x \in V_{H \triangleright C} \cup O_H$, then π has integrity for all H, C .*

7 Compositional Type Verification in *Prelude*

The *Prelude* language [28] includes structured data and function definitions for defining composable protocol elements at a higher level of abstraction than *Overture*. The *Prelude* language is a *metalanguage* aka metaprogramming language, where an *Overture* protocol is the result of computation. In addition to these declarative benefits of *Prelude*, component definitions support compositional verification of larger protocols. Separate verification of well-designed components results in confidentiality and integrity properties to be recorded in their types, allowing for significant reduction of SMT verification in whole program analysis, as we illustrate with extended examples in Section 8 and in an extended version of this paper [29].

$\ell \in \text{Field}, y \in \text{EVar}, f \in \text{FName}$		
$e ::= \nu \mid \mathbf{r}[e] \mid \mathbf{s}[e] \mid \mathbf{m}[e] \mid \mathbf{p}[e] \mid \mathbf{out} \mid e \text{ binop } e \mid y \mid$	$e.\ell \mid e\mathbf{0}e \mid \mathbf{let } y = e \text{ in } e \mid f(e, \dots, e) \mid \{\ell = e; \dots; \ell = e\}$	<i>expressions</i>
$\mathbf{c} ::= e := e \mid f(e, \dots, e) \mid \mathbf{let } y = e \text{ in } \mathbf{c} \mid \mathbf{c}; \mathbf{c}$	$\mathbf{assert}(e = e)\mathbf{0}e \mid \mathbf{m}[e]\mathbf{0}e \text{ as } \check{\phi}$	<i>instructions</i>
$\text{binop} ::= + \mid - \mid * \mid ++$		
$\nu ::= w \mid \iota \mid \varepsilon \mid x \mid \{\ell = \nu; \dots; \ell = \nu\}$		<i>values</i>
$\text{fn} ::= f(y, \dots, y)\{e\} \mid f(y, \dots, y)\{\mathbf{c}\}$		<i>functions</i>

Fig. 9. *Prelude* syntax.

7.1 Syntax and Semantics

The syntax of *Prelude* is defined in Figure 9. It includes a syntax of values ν that include client ids ι , identifier strings w , expressions ε in field $\mathbb{F}_p$, record values, and *Overture* variables x . *Prelude* expression forms allow dynamic construction of these values. *Prelude instruction* forms allow dynamic construction

of *Overture* protocols π that incorporate expression evaluation. The syntax also supports definitions of functions that compute values ν and, as a distinct form, functions that compute protocols π . Formally, we consider a complete metaprogram to include both a codebase and a “main” program that uses the codebase. We disallow recursion, mainly to guarantee decidability of type dependence (Section 7.2).

Definition 7.1. A codebase $\mathcal{C}$ is a list of function declarations. We write $\mathcal{C}(f) = y_1, \dots, y_n, e$ if $f(y_1, \dots, y_n)\{e\} \in \mathcal{C}$, and we write $\mathcal{C}(f) = y_1, \dots, y_n, \mathbf{c}$ if $f(y_1, \dots, y_n)\{\mathbf{c}\} \in \mathcal{C}$.

$$\begin{array}{c}
\frac{e_1 \Rightarrow \nu \quad e_2[\nu/y] \Rightarrow \nu'}{\text{let } y = e_1 \text{ in } e_2 \Rightarrow \nu'} \qquad \frac{e_1 \Rightarrow \varepsilon \quad e_2 \Rightarrow \iota}{e_1 @ e_2 \Rightarrow \varepsilon @ \iota} \\
\\
\frac{\mathcal{C}(f) = y_1, \dots, y_n, e \quad e_1 \Rightarrow \nu_1 \cdots e_n \Rightarrow \nu_n \quad e[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow \nu}{f(e_1, \dots, e_n) \Rightarrow \nu} \\
\\
\frac{e_1 \Rightarrow \nu_1 \quad \cdots \quad e_n \Rightarrow \nu_n}{\{\ell_1 = e_1; \dots; \ell_n = e_n\} \Rightarrow \{\ell_1 = \nu_1; \dots; \ell_n = \nu_n\}} \qquad \frac{e \Rightarrow \{\dots; \ell = \nu; \dots\}}{e.\ell \Rightarrow \nu} \\
\\
\frac{e_1 \Rightarrow w_1 \quad e_2 \Rightarrow w_2}{e_1 ++ e_2 \Rightarrow w_1 w_2} \qquad \frac{e \Rightarrow w}{\mathbf{m}[e] \Rightarrow \mathbf{m}[w]} \qquad \frac{e_1 \Rightarrow \varepsilon_1 \quad e_2 \Rightarrow \varepsilon_2}{e_1 + e_2 \Rightarrow \varepsilon_1 + \varepsilon_2}
\end{array}$$

Fig. 10. Semantics of *Prelude* expressions.

We define a big-step evaluation relation $\Rightarrow$ in Figures 10 and 11 for expressions and instructions, respectively. In this definition we write $e[\nu/y]$ and $\mathbf{c}[\nu/y]$ to denote the substitution of ν for free occurrences of y in e or $\mathbf{c}$ respectively. The rules are mostly standard. Note that we do not include an evaluation rule for the form $\mathbf{m}[e] @ e \text{ as } \check{\phi}$ which is a type annotation that we assume is erased from programs prior to evaluation. We defer discussion of this form, as well as the syntactic category $\check{\phi}$, to the next Section.

$$\begin{array}{c}
\frac{e_1 \Rightarrow x \quad e_2 \Rightarrow \varepsilon @ \iota}{e_1 := e_2 \Rightarrow x := \varepsilon @ \iota} \qquad \frac{e_1 \Rightarrow \varepsilon_1 \quad e_2 \Rightarrow \varepsilon_2 \quad e_3 \Rightarrow \iota}{\text{assert}(e_1 = e_2) @ e_3 \Rightarrow \text{assert}(\varepsilon_1 = \varepsilon_2) @ \iota} \\
\\
\frac{\mathcal{C}(f) = y_1, \dots, y_n, \mathbf{c} \quad e_1 \Rightarrow \nu_1 \cdots e_n \Rightarrow \nu_n \quad \mathbf{c}[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow \pi}{f(e_1, \dots, e_n) \Rightarrow \pi} \\
\\
\frac{e \Rightarrow \nu \quad \mathbf{c}[\nu/y] \Rightarrow \pi}{\text{let } y = e \text{ in } \mathbf{c} \Rightarrow \pi} \qquad \frac{e_1 \Rightarrow \pi_1 \quad e_2 \Rightarrow \pi_2}{e_1; e_2 \Rightarrow \pi_1; \pi_2}
\end{array}$$

Fig. 11. Semantics of *Prelude* instructions.

7.2 Dependent Hoare Type Theory

Our first goal in the *Prelude* type theory is to define an algorithmic system that is sound for both confidentiality and integrity typings as defined in Sections 5 and 6. Returning to the YGC secret encoding example in Section 5, note that the key equivalence of the `mux` expression with a one-time-pad encryption we observe there is not trivial. To ensure that this sort of equivalence can be picked up by the type system, we introduce an `as` annotation form that allows the programmer to provide the needed hint.

Consider the following `encode` function definition that generalizes this YGC encoding for any identifier y and sender, receiver pair s, r . The second line in the body is a hint that the message $m[y]@r$ can be considered equivalent to $\neg s[y]@s + r[y]@s$:

```

encode( $y, s, r$ ) {
   $m[y]@r := \text{mux}(s[y], \neg r[y], r[y])@s$ ;
   $m[y]@r$  as  $\neg s[y]@s + r[y]@s$ 
}

```

This hint can be validated using SMT, and then the syntactic structure of $\neg s[y]@s + r[y]@s$ allows its immediate interpretation as a one-time-pad encoding.

Our second goal in the *Prelude* type system is to minimize the amount of SMT solving needed for type verification. Returning to the `encode` example, type checking in our systems only requires the hint to be verified once, with the guarantee that `encode` can be applied anywhere without needing to re-verify the hint in application instances. To verify the hint in `encode`, we can just choose arbitrary “fresh” values w, ι_1, ι_2 for y, s , and r , evaluate `encode`(w, ι_1, ι_2) $\Rightarrow \pi$, and then verify:

$$[\pi] \models m[w]@_{\iota_2} \equiv \neg s[w]@_{\iota_1} + r[w]@_{\iota_1}$$

Since `encode` is closed, validity is guaranteed for any instance of y, s , and r .

We generalize these benefits of compositional verification by allowing pre- and postcondition annotations on *Prelude* functions. For example, consider a GMW-style, 2-party “and-gate” function `andgmw`(x, y, z). In this protocol, each party $\iota \in \{1, 2\}$ holds an additive secret share $m[x]@_{\iota}$ of values identified by x and y , and at the end of the protocol each hold a secret share $m[z]@_{\iota}$, where:

$$m[z]@_1 + m[z]@_2 \equiv (m[x]@_1 + m[x]@_2) * (m[y]@_1 + m[y]@_2)$$

We provide details of `andgmw` and other GMW protocol elements in Section 8. Our point now is that, similarly to the `encode` example, we can verify this postcondition once as a correctness property for `andgmw`, and then integrate instances of it into circuit correctness properties with the guarantee that each instance also holds for any `andgmw` gate. Since the program logic of `andgmw` is non-trivial, and typical circuits can use up to thousands of gates, this has significant practical benefits by greatly reducing SMT overhead in whole-program analysis.

$$\begin{array}{c}
\vdash x : (\emptyset, \{x\}) \quad \frac{\vdash \phi : (R, T) \quad \mathbf{r}[w]@l \notin R \quad \oplus \in \{+, -\}}{\vdash \phi \oplus \mathbf{r}[w]@l : (R \cup \{\mathbf{r}[w]@l\}, \{c(\mathbf{r}[w]@l, T)\})} \\
\\
\frac{\vdash \phi_1 : (R_1, T_1) \quad \vdash \phi_2 : (R_2, T_2) \quad \oplus \in \{+, -, *\}}{\vdash \phi_1 \oplus \phi_2 : (R_1; R_2, T_1 \cup T_2)}
\end{array}$$

Fig. 12. Algorithmic type judgements for *Overture*.

Overture expression type algorithm. A core element of *Prelude* type checking is type checking of *Overture* expressions. The integrity type system presented in Section 6 is already algorithmic and ready to use. But confidentiality typing presented in Section 5 is not syntax-directed due to the ENCODE rule. But as described above, by introducing hint annotations we can “coerce” any relevant expression form into the simplest equivalent syntactic form for one-time-pad encoding. Thus, in Figure 12 we present the algorithm for confidentiality type judgements, where we eliminate the need for integrated SMT solving by assuming this sort of casting.

The following Lemma establishes correctness of algorithmic confidentiality type checking, and makes explicit that SMT checking is eliminated in the judgement.

Lemma 7.1. *If $\vdash [\phi] : (R, T)$ then $R, E \vdash \phi : T$ for any E .*

Dependent Hoare types for instructions. Hoare-style types for instructions have the following form:

$$\{E_1\} \Gamma, R \cdot \Delta \{E_2\}$$

Here, E_1 and E_2 are the pre- and postconditions respectively, Γ is the confidentiality type environment of the protocol resulting from execution of the instruction, which is sound wrt confidentiality typing, R are the one-time-pads consumed in the confidentiality typing, and Δ is the sound integrity type environment.

To obtain a more complete type system, and to generalize function typings, we also introduce a form of type dependency, specifically dependence on *Prelude* expressions. Dependent Π types have the form:

$$\Pi y_1, \dots, y_n. \{ \check{E}_1 \} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{ \check{E}_2 \}$$

where $y_1, \dots, y_n$ range over values ν and each of $\check{E}_1$ etc. may contain expressions with free occurrences of $y_1, \dots, y_n$ —the syntax of these forms is in Figure 13. These Π types are assigned to functions and instantiated at application points.

For example, here is a valid typing of **encode**. It says that under any precondition, evaluating **encode** results in a cipher type for the encoded message $\mathbf{m}[x]@r$, which consumes the one-time-pad $\mathbf{r}[x]@s$, and the integrity of $\mathbf{m}[x]@r$ is determined by the security level of s . The postcondition expresses the key confidentiality property of $\mathbf{m}[x]@r$, but also may be practically useful for correctness

$$\begin{array}{ll}
\check{x} ::= \mathbf{r}[e] @ e \mid \mathbf{s}[e] @ e \mid \mathbf{m}[e] @ e \mid \mathbf{p}[e] \mid \mathbf{out} @ e & \check{t} ::= e \mid c(e, \check{T}) \\
\check{\phi} ::= \check{x} \mid \check{\phi} + \check{\phi} \mid \check{\phi} - \check{\phi} \mid \check{\phi} * \check{\phi} & \check{T} \in 2^{\check{t}} \\
\check{E} ::= \check{\phi} \equiv \check{\phi} \mid \check{E} \wedge \check{E} & \check{r} ::= \emptyset \mid \check{r}; e : \check{T} \\
\check{X} \in 2^{\check{x}} & \check{\Delta} ::= \emptyset \mid \check{\Delta}; e : e \cdot \check{V}
\end{array}$$

$$\frac{\check{\phi}_1 \Rightarrow \phi_1 \quad \check{\phi}_2 \Rightarrow \phi_2}{\check{\phi}_1 * \check{\phi}_2 \Rightarrow \phi_1 * \phi_2} \quad \frac{\check{\phi}_1 \Rightarrow \phi_1 \quad \check{\phi}_2 \Rightarrow \phi_2}{\check{\phi}_1 \equiv \check{\phi}_2 \Rightarrow \phi_1 \equiv \phi_2} \quad \frac{\check{E}_1 \Rightarrow E_1 \quad \check{E}_2 \Rightarrow E_2}{\check{E}_1 \wedge \check{E}_2 \Rightarrow E_1 \wedge E_2}$$

$$\frac{e \Rightarrow x \quad \check{T} \Rightarrow T}{c(e, \check{T}) \Rightarrow c(x, T)} \quad \frac{t_1 \Rightarrow t_1 \quad \dots \quad t_n \Rightarrow t_n}{\{\check{t}_1, \dots, \check{t}_n\} \Rightarrow \{t_1, \dots, t_n\}} \quad \frac{\check{r} \Rightarrow \Gamma \quad e \Rightarrow x \quad \check{T} \Rightarrow T}{\check{r}; e : \check{T} \Rightarrow \Gamma; x : T}$$

$$\frac{\check{\Delta} \Rightarrow \Delta \quad e_1 \Rightarrow x \quad e_2 \Rightarrow \iota \quad \check{V} \Rightarrow V}{\check{\Delta}; e_1 : e_2 \cdot \check{V} \Rightarrow \Delta; x : \iota \cdot V}$$

$$\frac{\check{E}_1 \Rightarrow E_1 \quad \check{r} \Rightarrow \check{R} \Rightarrow R \quad \check{\Delta} \Rightarrow \Delta \quad \check{E}_2 \Rightarrow E_2}{\{\check{E}_1\} \check{r}, \check{R} \cdot \check{\Delta} \{\check{E}_2\} \Rightarrow \{E_1\} \Gamma, R \cdot \Delta \{E_2\}}$$

Fig. 13. Evaluation of expressions within types and constraints.

properties since it is a simpler expression form than the **mux** form:

$$\begin{array}{l}
\mathbf{encode} : \Pi x, s, r. \{ \} \quad \mathbf{m}[x] @ r : c(\mathbf{r}[x] @ s, \mathbf{s}[x] @ s), \{ \mathbf{r}[x] @ s \} \cdot \\
\quad \mathbf{m}[x] @ r : s \cdot \{ \} \\
\{ \mathbf{m}[x] @ r \equiv \neg \mathbf{s}[y] @ s + \mathbf{r}[y] @ s \}
\end{array}$$

A key property of this example and our type system generally is that once the Π type is verified, typing any application of it requires verification of the precondition instance, but not the postcondition instance.

7.3 Algorithmic Type Validity in *Prelude*

We equate types up to evaluation of subexpressions as defined in Figure 13. Since expression evaluation is total we can just evaluate types of closed instructions, which are guaranteed to have closed types, to obtain Hoare typings for instructions. In Figure 14 we define the type derivation rules for *Prelude*. In the **MESG** rule we use algorithmic confidentiality and integrity type checking in a straightforward manner. In the **ENCODE** rule, we verify the hint given in the type annotation, and then use it for type checking. In the **APP** rule we verify preconditions of the given function type and instantiate the type and postconditions with the given expression arguments. The **SIG** rule we specify how to verify Π types for functions. Note that the values $\nu_1, \dots, \nu_n$ chosen to instantiate the function parameters are chosen “fresh”, meaning that any strings and client identifiers occurring as substrings in $\nu_1, \dots, \nu_n$ are unique and

$$\begin{array}{c}
\text{MESHG} \\
\frac{e_1 := e_2 \Rightarrow x := \varepsilon @ \iota \quad \Vdash \lfloor \varepsilon @ \iota \rfloor : (R, T) \quad \vdash_\iota \varepsilon : V}{\vdash e_1 := e_2 : \{E\} (x : T), R \cdot (x : \iota \cdot V) \{E \wedge x \equiv \lfloor \varepsilon @ \iota \rfloor\}} \\
\\
\text{ENCODE} \\
\frac{\mathbf{m}[e_1] @ e_2 \Rightarrow x \quad \check{\phi} \Rightarrow \phi \quad E \models x \equiv \phi \quad \Vdash \phi : (R, T)}{\vdash \mathbf{m}[e_1] @ e_2 \text{ as } \check{\phi} : \{E\} (x : T), R \cdot \emptyset \{E\}} \\
\\
\text{ASSERT} \\
\frac{\mathbf{assert}(e_1 = e_2) @ e_3 \Rightarrow \mathbf{assert}(\varepsilon_1 = \varepsilon_2) @ \iota \quad E \vdash \mathbf{assert}(\varepsilon_1 = \varepsilon_2) @ \iota : \Delta}{\vdash \mathbf{assert}(e_1 = e_2) @ e_3 : \{E\} \emptyset, \emptyset \cdot \Delta \{E\}} \\
\\
\text{APP} \\
\frac{f : \Pi y_1, \dots, y_n. \check{E}_1 \check{\Gamma}, \check{R} \cdot \check{\Delta} \{ \check{E}_2 \} \quad e_1 \Rightarrow \nu_1 \dots e_n \Rightarrow \nu_n \quad (\{ \check{E}_1 \} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{ \check{E}_2 \}) [\nu_1 / y_1] \dots [\nu_n / y_n] \Rightarrow \{E_1\} \Gamma, R \cdot \Delta \{E_2\} \quad E \models E_1}{\vdash f(e_1, \dots, e_n) : \{E\} \Gamma, R \cdot \Delta \{E \wedge E_2\}} \\
\\
\text{SEQ} \\
\frac{\vdash \pi_1 : \{E_1\} \Gamma_1, R_1 \cdot \Delta_1 \{E_2\} \quad \vdash \pi_2 : \{E_2\} \Gamma_2, R_2 \cdot \Delta_2 \{E_3\}}{\vdash \pi_1; \pi_2 : \{E_1\} \Gamma_1; \Gamma_2, R_1; R_2 \cdot \Delta_1; \Delta_2 \{E_3\}}
\end{array}$$

Fig. 14. *Prelude* type derivation rules for instructions.

not used in any program source code. This ensures generality of verification as demonstrated in an extended version of this paper [29].

We allow the programmer to specify a type signature sig that maps functions in $\mathcal{C}$ to valid dependent Hoare typings. Define:

Definition 7.2. sig is verified iff $f : \text{sig}(f)$ is valid for all $f \in \text{dom}(\text{sig})$.

In practice, to define sig the programmer only needs to specify pre- and post-conditions, as we illustrate in Section 8 and [29], since the rest of the type is reconstructed by type checking.

The following Theorem establishes our main result, that the *Prelude* type checking is sound with respect to both confidentiality and integrity typings. Since these typings imply security hyperproperties of confidentiality and integrity, *Prelude* type checking implicitly enforces them. Proof details are given [29].

Theorem 7.1 (Soundness of *Prelude* type checking). *Given preprocessing predicate E_{pre} , program $\mathbf{c}$, and verified sig , if the judgement:*

$$\vdash \mathbf{c} : \{E_{\text{pre}}\} \Gamma, R \cdot \Delta \{E\}$$

is derivable then $\mathbf{c} \Rightarrow \pi$ and:

1. $R, E \vdash E_{\text{pre}} \wedge \lfloor \pi \rfloor : \Gamma$ is valid.
2. $E \vdash \pi : \Delta$ is valid.

FN

$$\frac{\begin{array}{l} \mathcal{C}(f) = y_1, \dots, y_n, \mathbf{c} \quad \vdash \mathbf{c}[\nu_1/y_1] \cdots [\nu_n/y_n] : \{E_1\} \quad \Gamma, R \cdot \Delta \{E_2\} \\ \text{fresh}(\nu_1, \dots, \nu_n) \quad (\{\check{E}_1\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}_2\})[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow \{E_1\} \quad \Gamma, R \cdot \Delta \{E_2\} \end{array}}{f : \Pi y_1, \dots, y_n. \{\check{E}_1\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}_2\}}$$

FNPRE

$$\frac{\begin{array}{l} f : \Pi y_1, \dots, y_n. \{\check{E}\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}_2\} \quad \text{precond}(f) = \check{E}_1 \quad \text{fresh}(\nu_1, \dots, \nu_n) \\ \check{E}[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow E \quad \check{E}_1[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow E_1 \quad E_1 \models E \end{array}}{f : \Pi y_1, \dots, y_n. \{\check{E}_1\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}_2\}}$$

FNPOST

$$\frac{\begin{array}{l} f : \Pi y_1, \dots, y_n. \{\check{E}_1\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}\} \quad \text{postcond}(f) = \check{E}_2 \quad \text{fresh}(\nu_1, \dots, \nu_n) \\ \check{E}[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow E \quad \check{E}_2[\nu_1/y_1] \cdots [\nu_n/y_n] \Rightarrow E_2 \quad E \models E_2 \end{array}}{f : \Pi y_1, \dots, y_n. \{\check{E}_1\} \check{\Gamma}, \check{R} \cdot \check{\Delta} \{\check{E}_2\}}$$

Fig. 15. *Prelude* type derivation rules for function definitions.

8 Examples and Evaluation

In Section 4 we gave the example of 3-party additive sharing for any field $\mathbb{F}_p$. In Sections 6 and 7.2 we discussed the example of YGC input encoding. Now we consider examples that illustrate the versatility of the ENCODE typing rule for confidentiality and the use of postconditions for correctness. In an extended version of this paper we also exemplify the use of both pre- and postconditions for integrity [29].

To evaluate our system especially regarding performance, using the Satisfiability Modulo Fields theory in cvc5 on an Apple M1 processor, we have verified correctness of additive sharing protocols in fields up to size $\mathbb{F}_{2^{31}-1}$ (an approximation of 32-bit integers) which takes $\sim 7\text{ms}$ to compute for 3 parties and not significantly more than for, e.g., $\mathbb{F}_2$. We have also verified the semi-homomorphic encryption properties leveraged in the BDOZ circuit library functions defined in [29] in fields up to size $\mathbb{F}_{2^{31}-1}$ which take $\sim 8\text{ms}$ for the property used for the postcondition of the `mult` gate. This is categorically better performance than can be achieved using previous brute-force methods in $\mathbb{F}_2$ [28].

GMW Input Encoding and And Gate. In the 2-party GMW protocol [11], another garbled binary circuit protocol, values are encrypted in a manner similar to using secret sharing as described in Sections 4.2 and 6. In our definition of GMW we use the convention that shared values $\langle \mathbf{m}[w] \rangle$ are identified by strings w and encoded as shares $\mathbf{m}[w]@1$ and $\mathbf{m}[w]@2$. Ciphertexts reflect the confidentiality of GMW input encodings as defined in the `encodegmw` function in Figure 16. No programmer annotation is needed given the syntax of the function body. More interestingly, the `andgmw` gate definition shows how a programmer hint can express the relevant confidentiality property of the output share $\mathbf{m}[z]@2$ on client 2, using the `as` casting. The non-trivial equivalence can be verified by SMT once during

```

encodegmw(n, i1, i2) {
  m[n]@i2 := (s[n] + r[n])@i1;
  m[n]@i1 := r[n]@i1
}

andtablegmw(x, y, z) {
  let r11 = r[z] + (m[x] + 1) * (m[y] + 1) in
  let r10 = r[z] + (m[x] + 1) * (m[y] + 0) in
  let r01 = r[z] + (m[x] + 0) * (m[y] + 1) in
  let r00 = r[z] + (m[x] + 0) * (m[y] + 0) in
  { row1 = r00; row2 = r01; row3 = r10; row4 = r11 }
}

andgmw(z, x, y) {
  let table = andtablegmw(x,y,z) in
  m[z]@2 := OT4(m[x], m[y], table, 2, 1);
  m[z]@2 as ~(m[x]@1 + m[x]@2) * (m[y]@1 + m[y]@2)) + r[z]@1;
  m[z]@1 := r[z]@1
}
post:
{ m[z]@1 + m[z]@2 == (m[x]@1 + m[x]@2) * (m[y]@1 + m[y]@2) }

```

Fig. 16. 2-party GMW circuit library.

verification, and subsequently confidentiality is expressed in the dependent type of the function (as for `encode`).

Note also that the `andgmw` function is decorated with a postcondition that expresses the correctness property of the function. Although not strictly necessary for confidentiality, this annotation can at least help eliminate bugs, and also can be used compositionally for whole-program correctness properties. As for confidentiality hints, this postcondition needs to be verified only once at the point of definition and subsequently is guaranteed to hold for any application.

9 Conclusion and Future Work

In this paper we have defined confidentiality and integrity type systems for the *Prelude/Overture* language model. These type systems resemble previous security type systems in probabilistic settings, but their accuracy is boosted by the integral use of SMT verification. Our work significantly improves automation, scalability, and flexibility of static analysis in the *Prelude/Overture* as compared to prior work.

We see several promising avenues for future work. Concurrency will be important to capture common MPC idioms such as commitment and circuit optimizations and to consider the UC security model [2, 11]. We also hope to improve our type system to enforce the declassification bounds determined by ideal functionalities, beyond gradual release.

References

1. A probabilistic separation logic. *Proc. ACM Program. Lang.* **4**(POPL) (dec 2019). <https://doi.org/10.1145/3371123>
2. Acay, C., Gancher, J., Recto, R., Myers, A.: Secure synthesis of distributed cryptographic applications. In: 2024 IEEE 37th Computer Security Foundations Symposium (CSF). pp. 315–330. IEEE Computer Society, Los Alamitos, CA, USA (jul 2024). <https://doi.org/10.1109/CSF61375.2024.00021>, <https://doi.ieeecomputersociety.org/10.1109/CSF61375.2024.00021>
3. Acay, C., Recto, R., Gancher, J., Myers, A.C., Shi, E.: Viaduct: An extensible, optimizing compiler for secure distributed programs. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. pp. 740–755. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021)<https://doi.org/10.1145/3453483.3454074>
4. Almeida, J.B., Barbosa, M., Barthe, G., Pacheco, H., Pereira, V., Portela, B.: Enforcing ideal-world leakage bounds in real-world secret sharing mpc frameworks. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 132–146. IEEE (2018)
5. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) *Advances in Cryptology – EUROCRYPT 2011*. pp. 169–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. pp. 53–65. PLAS’14, Association for Computing Machinery, New York, NY, USA (2014). <https://doi.org/10.1145/2637113.2637119>
7. Damgård, I., Orlandi, C.: Multiparty computation for dishonest majority: From passive to active security at low cost. In: Rabin, T. (ed.) *Advances in Cryptology – CRYPTO 2010*. pp. 558–576. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) *Advances in Cryptology – CRYPTO 2012*. pp. 643–662. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
9. Darais, D., Sweet, I., Liu, C., Hicks, M.: A language for probabilistically oblivious computation. *Proceedings of the ACM on Programming Languages* **4**(POPL), 1–31 (2019)
10. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Blazy, S., Chechik, M. (eds.) *Verified Software. Theories, Tools, and Experiments*. pp. 56–72. Springer International Publishing, Cham (2016)
11. Evans, D., Kolesnikov, V., Rosulek, M., et al.: A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* **2**(2–3), 70–246 (2018)
12. Gao, S., Peng, Z., Tan, F., Zheng, Y., Xiao, B.: Symmeproof: Compact zero-knowledge argument for blockchain confidential transactions. *IEEE Transactions on Dependable and Secure Computing* (2022)
13. Haagh, H., Karbyshev, A., Oechsner, S., Spitters, B., Strub, P.: Computer-aided proofs for multiparty computation with active security. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 119–131. IEEE Computer Society

- ety, Los Alamitos, CA, USA (jul 2018). <https://doi.org/10.1109/CSF.2018.00016>, <https://doi.ieeecomputersociety.org/10.1109/CSF.2018.00016>
14. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. *SIAM Journal on Computing* **39**(3), 1121–1152 (2009)
 15. Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M., van der Maaten, L.: Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems* **34**, 4961–4973 (2021)
 16. Koch, K., Krenn, S., Pellegrino, D., Ramacher, S.: Privacy-preserving analytics for data markets using mpc. In: *IFIP International Summer School on Privacy and Identity Management*, pp. 226–246. Springer (2020)
 17. Li, J.M., Ahmed, A., Holtzen, S.: Lilac: a modal separation logic for conditional probability. *Proceedings of the ACM on Programming Languages* **7**(PLDI), 148–171 (2023)
 18. Li, X., Dowsley, R., De Cock, M.: Privacy-preserving feature selection with secure multiparty computation. In: *International Conference on Machine Learning*. pp. 6326–6336. PMLR (2021)
 19. Lindell, Y.: *How to Simulate It – A Tutorial on the Simulation Proof Technique*, pp. 277–346. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-57048-8_6, https://doi.org/10.1007/978-3-319-57048-8_6
 20. Liu, J., Tian, Y., Zhou, Y., Xiao, Y., Ansari, N.: Privacy preserving distributed data mining based on secure multi-party computation. *Computer Communications* **153**, 208–216 (2020)
 21. Lu, D., Yurek, T., Kulshreshtha, S., Govind, R., Kate, A., Miller, A.: Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. pp. 887–903 (2019)
 22. Mitchell, J.C., Sharma, R., Stefan, D., Zimmerman, J.: Information-flow control for programming on encrypted data. In: *2012 IEEE 25th Computer Security Foundations Symposium*. pp. 45–60 (2012). <https://doi.org/10.1109/CSF.2012.30>
 23. Orsini, E.: Efficient, actively secure MPC with a dishonest majority: A survey. In: Bajard, J.C., Topuzoğlu, A. (eds.) *Arithmetic of Finite Fields*. pp. 42–71. Springer International Publishing, Cham (2021)
 24. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields. In: Enea, C., Lal, A. (eds.) *Computer Aided Verification*. pp. 163–186. Springer Nature Switzerland, Cham (2023)
 25. Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: *2014 IEEE Symposium on Security and Privacy*. pp. 655–670. IEEE (2014)
 26. Rastogi, A., Swamy, N., Hicks, M.: Wys*: A DSL for verified secure multi-party computations. In: Nielson, F., Sands, D. (eds.) *8th International Conference on Principles of Security and Trust (POST)*. *Lecture Notes in Computer Science*, vol. 11426, pp. 99–122. Springer (2019). https://doi.org/10.1007/978-3-030-17138-4_5
 27. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. *J. Comput. Secur.* **17**(5), 517–548 (oct 2009)
 28. Skalka, C., Near, J.P.: Language-based security for low-level MPC. In: *Proceedings of the ACM Conference on Principles and Practice of Declarative Programming* (2024)
 29. Skalka, C., Near, J.P.: Smt-boosted security types for low-level mpc (2025). <https://arxiv.org/abs/2501.17824>

30. Tomaz, A.E.B., Do Nascimento, J.C., Hafid, A.S., De Souza, J.N.: Preserving privacy in mobile health systems using non-interactive zero-knowledge proof and blockchain. *IEEE access* **8**, 204441–204458 (2020)
31. Ye, Q., Delaware, B.: Oblivious algebraic data types. *Proc. ACM Program. Lang.* **6**(POPL) (jan 2022). <https://doi.org/10.1145/3498713>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Context-Dependent Effects in Guarded Interaction Trees

Sergei Stepanenko¹ , Emma Nardino², Dan Frumin³ , Amin Timany¹ ,
and Lars Birkedal¹

¹ Aarhus University, Aarhus, Denmark

{sergei.stepanenko,timany,birkedal}@cs.au.dk

² Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP, UMR 5668, F-69342, Lyon
CEDEX 07, France

emma.nardino@ens-lyon.fr

³ University of Groningen, Groningen, The Netherlands
d.frumin@rug.nl

Abstract. Guarded Interaction Trees are a structure and a fully formalized framework for representing higher-order computations with higher-order effects in Coq. We present an extension of Guarded Interaction Trees to support formal reasoning about context-dependent effects. That is, effects whose behaviors depend on the evaluation context, e.g., `call/cc`, `shift` and `reset`. Using and reasoning about such effects is challenging since certain compositionality principles no longer hold in the presence of such effects. For example, the so-called “bind rule” in modern program logics (which allows one to reason modularly about a term inside a context) is no longer valid. The goal of our extension is to support representation and reasoning about context-dependent effects in the most painless way possible. To that end, our extension is conservative: the reasoning principles (and the Coq implementation) for context-independent effects remain the same. We show that our implementation of context-dependent effects is viable and powerful. We use it to give direct-style denotational semantics for higher-order programming languages with `call/cc` and with delimited continuations. We extend the program logic for Guarded Interaction Trees to account for context-dependent effects, and we use the program logic to prove that the denotational semantics is adequate with respect to the operational semantics. This is achieved by constructing logical relations between syntax and semantics inside the program logic. Additionally, we retain the ability to combine multiple effects in a modular way, which we demonstrate by showing type soundness for safe interoperability of a programming language with delimited continuations and a programming language with higher-order store.

Keywords: Coq · Iris · denotational semantics · logical relations · control flow operators · continuations · delimited continuations

1 Introduction

Despite a lot of recent progress, representing and reasoning about programming languages in proof assistants, such as Coq, is still considered a major challenge.

The design space is wide and many approaches have been considered. Recently, research on a novel point in the design space was initiated with the introduction of Interaction Trees [34], or ITrees for short. ITrees were introduced to simplify representation and reasoning about possibly non-terminating programs with side effects in Coq. In a sense, ITrees provide a target for denotational semantics of programming languages, which allows one to abstract from syntactic details often found in models based on operational semantics. ITrees specifically allow one to easily represent and reason about various effects and their combinations in a modular way. A wide range of subsequent applications of ITrees (see, e.g., [18,36,35,25], among others) show that they indeed work very well for representing and reasoning about first-order programs with first-order effects. As part of the trade-offs, ITrees could not support higher-order representations and higher-order effects. To address this challenge, Guarded Interaction Trees (or GITrees for short) were introduced [13].

While GITrees support *higher-order effects*, in particular, the challenging case of higher-order store, they are limited to effects that do not alter the control flow of the program (more specifically, the continuation). As a consequence, one cannot use GITrees to give direct-style denotational semantics of programming languages with context-dependent effects such as `call/cc`, exceptions, or delimited continuations. It is of course possible to give semantics to, e.g., `call/cc` using a CPS translation, but this would require a global transformation which complicates representation and reasoning, especially in combination with other effects present. In this paper we extend GITrees to support direct-style representation and reasoning about higher-order programs with higher-order context-dependent effects in Coq, and evaluate its modularity.

We want to stress that our extension to context-dependent effects is not only theoretically interesting, but also important for scalability, since many real mainstream programming languages include context-dependent effects. Indeed, exceptions are now a standard feature in many languages, and while other context-dependent effects such as delimited continuations are not as widespread in mainstream programming languages, they are present in the core calculi used for some such languages: for example, the Glasgow Haskell Compiler core language was recently extended with delimited continuations to support the introduction of effect systems, which can be efficiently developed on top of delimited continuations [16]. Moreover, effect handlers, which rely on control flow operators, have recently been introduced in OCaml 5.0, and as a design feature in Helium [7], Koka [21,20], and other languages. Note that these languages do not only include forms of delimited continuations, but also other effects, which underscores the importance of considering delimited control in combination with other effects.

Overview of technical development and key challenges. Similarly to how ITrees are defined as a coinductive type in Coq, GITrees are defined as a guarded recursive type (this is to support function spaces in GITrees; in the presence of function spaces there are negative occurrences of the recursive type and hence one cannot simply define the type of GITrees using coinduction). Coq does not

directly support guarded recursive types, so GITrees are defined using a fragment of guarded type theory implemented in Coq, as part of the Iris framework [14]. To work efficiently with GITrees we make use of other Iris features like separation logic and Iris Proof Mode [19], which we use to define custom program logics for different (combinations of) effects. This enables us to reason about GITrees smoothly in the Iris logic in Coq, in much the same way as one works directly in Coq. We recall the precise definition of GITrees in [Section 2](#).

Broadly speaking, GITrees model effects in the following way. The type of GITrees is parameterized over a set of effectful operations. Each operation is given meaning by a *reifier* function, using a form of state monad. From this, we define the *reduction* relation of GITrees, which gives semantics to computations represented by GITrees.

To support context-dependent effects, we extend ([Section 3](#)) the notion of a reifier so that reification of effects can also depend on the context; technically, the reifier operation becomes parameterized by a suitable GITrees continuation. This extension allows us to give semantics to context-dependent effects, but it comes at a price. In particular, following the change in semantics, we need to reformulate the program logic for GITrees: in the presence of context-dependent effects (like `call/cc`), the so-called “bind”-rule becomes unsound. Of course, we do not want this reformulation to complicate reasoning about computations that do *not* include context-dependent effects. To that end, we parameterize the GITrees (and the program logic) by a flag, which allows us to recover the original proof rules and make sure that all of the original GITrees framework still works with our extension.

To motivate the extension to context-dependent effects, we give direct-style denotational semantics to a higher-order programming language $\lambda_{\text{call/cc}}$ with `call/cc` ([Section 3](#)). Furthermore, we use the derived program logic to construct a logical relation between the denotational and the operational semantics to prove computational adequacy of our model.

Our main interest, however, lies in the treatment of delimited continuations. In [Section 4](#) we show how to represent delimited continuations as effects in GITrees, and we use them to define a novel denotational semantics for a programming language with `shift` and `reset` operators. We prove that our denotational semantics is sound with respect to the operational semantics (given by an extension of the CEK abstract machine). We additionally use the program logic to define a logical relation, and prove computational adequacy and *semantic* type soundness. We recall that semantic type soundness is interesting because it allows one to combine syntactically well-formed programs with syntactically ill-typed, but semantically well-behaved programs [30].

As we mentioned, it is important to consider delimited continuations not only on their own, but in combinations with other effects. And indeed, one of the key points of ITrees, and therefore also of GITrees, is that they support reasoning about effecting and language interoperability by establishing a common unifying semantic framework. In this paper, we consider (in [Section 5](#)) an example of such interaction: we show a type-safe embedding of λ_{delim} with delimited continuations

into a language λ_{embed} with higher-order store. We allow λ_{delim} expressions to be embedded into λ_{embed} by surrounding them by simple glue code, and use a type system to ensure type safety of the combined language. To define the semantics of the combined language we rely on the modularity of GITrees, and combine reifiers for delimited continuations with reifiers for higher-order store. We prove type safety of the combined language by constructing a logical relation and use the program logic both to define the logical relation and to verify the glue code between the two languages. The type system for the combined language naturally requires that the embedded code is well-typed according to the type system for λ_{delim} and thus we can rely on the type soundness of λ_{delim} (proved in the earlier Section 4) when proving type safety for the combined language. At the end of Section 5, we give an example of how to verify a more involved interaction of effects, albeit without the type system.

Summary of Contributions. In summary, we present:

- A conservative (with respect to the old results) extension to GITrees for representing and reasoning about context-dependent effects (Section 3).
- A sound and adequate model of a calculus with `call/cc` and `throw`, implemented in a direct style (Section 3.3).
- A sound and adequate model of a calculus with delimited continuations, with operations `shift` and `reset`, implemented in a direct style (Section 4).
- A type system for interoperability between a programming language with delimited continuations and a programming language with higher-order store, with a semantic type safety proof (Section 5).

All results in the paper have been formalized in Coq as a modification to the GITrees library and the previously proved results have been ported to our extension. We conclude and discuss related work in Section 6. Before we go on with the main part of the paper, we recall some background material on GITrees.

2 Guarded Interaction Trees

In this section we provide an introduction to guarded interaction trees. Our treatment is brief, and we refer the reader to the original paper for details [13].

Iris and Guarded Type Theory. Guarded Interaction trees (GITrees) are defined in Iris logic. Here we briefly touch Iris, and refer the reader to the literature on Iris [15] and guarded type theory [8] for more in-depth details. Iris is a separation logic framework built on top of a model of *guarded type theory*, the main use of which is to solve recursive equations and define guarded recursive types, such as the type of GITrees described below. Moreover, Iris has a specialized proof mode [19], implemented in Coq. This allows the users of Iris to carry out formal reasoning in separation logic as if they are proving things normally Coq, as we have done in the formalization of this work. For this reason, in the paper we will

$$\begin{aligned}
\tau &::= \text{iProp} \mid 0 \mid \mathbf{1} \mid \mathbb{B} \mid \text{Nat} \mid \tau + \tau \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \blacktriangleright \tau \mid I \mid \sum_{i \in I} \tau_i \mid \prod_{i \in I} \tau_i \mid \dots \\
t &::= x \mid F(t_1, \dots, t_n) \mid \text{abort } t \mid () \mid (t, t) \mid \pi_i t \mid \lambda x : \tau. t \mid \\
&\quad \text{inj}_i t \mid \text{match } t \text{ with } \overline{\text{inj}_i x.t} \text{ end} \mid \text{next}(t) \mid \text{fix}_\tau \mid \dots \\
P &::= \text{False} \mid t =_\tau t \mid P \vee P \mid P \rightarrow P \mid \forall x : \tau. P \mid P * P \mid P \multimap P \mid \Box P \mid \boxed{P} \mid \triangleright P \mid \dots
\end{aligned}$$
Fig. 1. Grammar for the Iris base logic.
$$\begin{aligned}
&\text{guarded type } \mathbf{IT}_E(A) = \text{Ret} : A \rightarrow \mathbf{IT}_E(A) \\
&\mid \text{Fun} : \blacktriangleright(\mathbf{IT}_E(A) \rightarrow \mathbf{IT}_E(A)) \rightarrow \mathbf{IT}_E(A) \\
&\mid \text{Err} : \text{Error} \rightarrow \mathbf{IT}_E(A) \\
&\mid \text{Tau} : \blacktriangleright \mathbf{IT}_E(A) \rightarrow \mathbf{IT}_E(A) \\
&\mid \text{Vis} : \prod_{i \in I} (\text{Ins}_i(\mathbf{IT}_E(A)) \times (\text{Outs}_i(\mathbf{IT}_E(A)) \rightarrow \blacktriangleright \mathbf{IT}_E(A))) \rightarrow \mathbf{IT}_E(A)
\end{aligned}$$
Fig. 2. Guarded datatype of interaction trees.

work with Iris and its type theory informally. Still, we need to say a few things about the foundations.

The syntax of Iris, shown in [Figure 1](#), contains types, terms, and propositions. The grammar is standard for higher-order logic, with the exception of the guarded types fragment, and separation logic connectives. The type of propositions is denoted `iProp`. The *guarded* part of guarded type theory is the “later” modality $\blacktriangleright$. Intuitively, we view all types as indexed by a natural number, where τ_n contains elements of τ “at time” n . Then $\blacktriangleright \tau$ contains elements of τ at a later time; that is, $(\blacktriangleright \tau)_n = \tau_{n-1}$. There is an embedding $\text{next} : \tau \rightarrow \blacktriangleright \tau$, and there is a *guarded* fixed point combinator $\text{fix}_\tau : (\blacktriangleright \tau \rightarrow \tau) \rightarrow \tau$, similar to the unguarded version in PCF. We can also lift functions to $\blacktriangleright$: given $f : A \rightarrow B$, we have $\blacktriangleright f : \blacktriangleright A \rightarrow \blacktriangleright B$.

For the proposition, Iris contains the usual separation logic connectives, and the two modalities: “later” $\triangleright$ and “persistently” $\Box$. The propositional $\triangleright$ modality reflects the type-level later modality $\blacktriangleright$ on the level of propositions, as justified by the following rule: $\triangleright(\alpha =_\tau \beta) \dashv\vdash \text{next}(\alpha) =_{\blacktriangleright \tau} \text{next}(\beta)$. The persistence modality $\Box P$ states that the proposition P is available without claiming any resources (as it normally is the case in separation logic); crucially it makes the proposition duplicable: $\Box P \vdash (\Box P) * (\Box P)$. An example of a persistent proposition is the invariant proposition $\boxed{P}$, which satisfies $\boxed{P} \vdash \Box \boxed{P}$.

Guarded Interaction Trees. Guarded recursive datatypes are datatypes obtained from recursive equations of the form $X = F(\blacktriangleright X)$. In other words, guarded recursive datatypes are similar to the regular datatypes you see in normal programming languages, but every recursive occurrence of the type must be

guarded by the $\blacktriangleright$ modality. The datatype we are concerned with here is the type of GITrees , shown in Figure 2. It is parameterized over two types: the ground type A and the effect signature E (more on it below).

Guarded Interaction Trees represent computational trees in which the leaves are of the ground type ($\text{Ret}(a)$), error states ($\text{Err}(e)$), and functions ($\text{Fun}(f)$). The leaves $\text{Ret}(a)$ and $\text{Fun}(f)$ are also called *values*, and we write $\mathbf{IT}_E^v(A)$ for the type of $\mathbf{IT}_E(A)$ -values.

The nodes of the computation trees are of the two kinds. The first one is a “silent step” constructor $\text{Tau}(\alpha)$. It represents an unobservable internal step of the computation. For convenience, we use the function $\text{Tick} \triangleq \text{Tau} \circ \text{next} : \mathbf{IT}_E(A) \rightarrow \mathbf{IT}_E(A)$ that “delays” its argument. This function satisfies the following equation: $\text{Tick}(\alpha) = \text{Tick}(\beta) \dashv\vdash \triangleright(\alpha = \beta)$.

The second kind of nodes are effects given by $\text{Vis}_i(x, k)$. The parameters I , Ins and Outs are part of the effect signature E . The set I is the set of *names* of operations. The *arities* of an operation $i \in I$ are given by functors Ins_i and Outs_i . Let us give an example.

Consider the following signature for store effects. The signature E_{state} consists of effects $\{\text{write}, \text{read}, \text{alloc}\}$ with the following input/output arities:

$$\begin{array}{lll} \text{Ins}_{\text{write}}(X) \triangleq \text{Loc} \times \blacktriangleright X & \text{Ins}_{\text{read}}(X) \triangleq \text{Loc} & \text{Ins}_{\text{alloc}}(X) \triangleq \blacktriangleright X \\ \text{Outs}_{\text{write}}(X) \triangleq 1 & \text{Outs}_{\text{read}}(X) \triangleq \blacktriangleright X & \text{Outs}_{\text{alloc}}(X) \triangleq \text{Loc} \end{array}$$

For example, **write** expect a location and a new GITree as its input, and simply returns the unit value as an output. We write $\text{Vis}_{\text{write}}((\ell, \alpha), \lambda\_. \beta)$ for the computation that invokes the **write** effect with arguments ℓ and α , waits for it to return, and proceeds as β . Thus, the first argument for Vis_i is the input, and the second one is the continuation dependent on the output. This continuation determines the branching in (G)ITrees.

For effects like above, it is usually convenient to provide wrappers:

$$\begin{aligned} \text{Alloc}(\alpha : \mathbf{IT}, k : \text{Loc} \rightarrow \mathbf{IT}) &\triangleq \text{Vis}_{\text{alloc}}(\text{next}(\alpha), \text{next} \circ k) \\ \text{Read}(\ell : \text{Loc}) &\triangleq \text{Vis}_{\text{read}}(\ell, \lambda x.x) \\ \text{Write}(\ell : \text{Loc}, \alpha : \mathbf{IT}) &\triangleq \text{Vis}_{\text{write}}((\ell, \text{next}(\alpha)), \lambda x.\text{next}(\text{Ret}(\text{inj}))) \end{aligned}$$

When the signature and the return type are clear from the context, we simply write $\mathbf{IT}$ and $\mathbf{IT}^v$ for the GITrees and GITree -values.

Equational theory. GITrees come with a number of operations (defined using the recursion principle) that are used for writing and composing computations. Here we list some of those operations which we will be using. The function $\text{get\_val}(\alpha, f : \mathbf{IT}^v \rightarrow \mathbf{IT})$ are used for sequencing computations, and its corresponding equations are shown in Figure 3. Intuitively, $\text{get\_val}(\alpha, f)$ first tries to compute α to a value (a $\text{Ret}(a)$ or a $\text{Fun}(g)$), and then calls f on that value. Similarly, $\text{get\_fun}(\alpha : \mathbf{IT}, f : \blacktriangleright(\mathbf{IT} \rightarrow \mathbf{IT}) \rightarrow \mathbf{IT})$ and $\text{get\_ret}(\alpha : \mathbf{IT}_E(A), f : A \rightarrow \mathbf{IT}_E(A))$ first compute α to a value; if that value is a function $\text{Fun}(g)$ (resp., $\text{Ret}(a)$), then it proceeds with $f(g)$ (resp., $f(a)$). Otherwise it results in a runtime error.

$$\begin{aligned}
\text{get\_val}(\text{Ret}(a), f) &= f(\text{Ret}(a)) & \text{get\_val}(\text{Tau}(t), f) &= \text{Tau}(\blacktriangleright \text{get\_val}(t, f)) \\
\text{get\_val}(\text{Fun}(g), f) &= f(\text{Fun}(g)) & \text{get\_val}(\text{Tick}(\alpha), f) &= \text{Tick}(\text{get\_val}(\alpha, f)) \\
\text{get\_val}(\text{Err}(e), f) &= \text{Err}(e) & \text{get\_val}(\text{Vis}_i(x, k), f) &= \text{Vis}_i(x, \blacktriangleright \text{get\_val}(-, f) \circ k)
\end{aligned}$$

Fig. 3. Example function on Guarded Interaction Trees.

$$\begin{aligned}
r &: \prod_{i \in E} \text{Ins}_i(\mathbf{IT}_E) \times \text{State} \rightarrow \text{option}(\text{Outs}_i(\mathbf{IT}_E) \times \text{State}) \\
\frac{r_i(x, \sigma) = \text{Some}(y, \sigma') \quad k \ y = \text{next}(\beta)}{\text{reify}(\text{Vis}_i(x, k), \sigma) = (\text{Tick}(\beta), \sigma')} & \quad \frac{r_i(x, \sigma) = \text{None}}{\text{reify}(\text{Vis}_i(x, k), \sigma) = (\text{Err}(\text{RunTime}), \sigma)}
\end{aligned}$$

Fig. 4. Signature of reifiers and the reification function

Crucially, to work with higher-order computations, GITrees provide the “call-by-value” application $\alpha \bullet \beta$ satisfying the following equations:

$$\begin{aligned}
\alpha \bullet \text{Tick}(\beta) &= \text{Tick}(\alpha \bullet \beta) & \alpha \bullet \text{Vis}_i(x, k) &= \text{Vis}_i(x, \lambda y. \text{next}(\alpha) (\blacktriangleright \bullet) k \ y) \\
\text{Tick}(\alpha) \bullet \beta_v &= \text{Tick}(\alpha \bullet \beta_v) & \text{Vis}_i(x, k) \bullet \beta_v &= \text{Vis}_i(x, \lambda y. k \ y (\blacktriangleright \bullet) \text{next}(\beta_v)) \\
\text{Fun}(\text{next}(g)) \bullet \beta_v &= \text{Tick}(g(\beta_v)) & \alpha \bullet \beta &= \text{Err}(\text{RunTime}) \text{ in other cases}
\end{aligned}$$

where $-(\blacktriangleright \bullet)-$ is defined as the lifting of $-\bullet-$ to $\blacktriangleright \mathbf{IT}_E(A) \rightarrow \blacktriangleright \mathbf{IT}_E(A) \rightarrow \blacktriangleright \mathbf{IT}_E(A)$, and $\beta_v \in \mathbf{IT}_E^v(A)$ is either $\text{Ret}(a)$ or $\text{Fun}(g)$.

The application function $\alpha \bullet \beta$ simulates strict function application. It first tries to evaluate β to a value β_v . Then it tries to evaluate α to a function f . If it succeeds, then it invokes $f(\beta_v)$. If at any point it fails, application results in a runtime error.

For the often-used case of GITrees where the ground type includes natural numbers, we use the function $\text{NatOp} : (\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbf{IT} \rightarrow \mathbf{IT} \rightarrow \mathbf{IT}$ which lifts binary functions on natural numbers to binary functions on GITrees. That is, $\text{NatOp}_f(\alpha, \beta)$ first evaluate GITrees β and α to values. If those values are natural numbers, then it computes f of those numbers and returns the result as a GITree. Otherwise, it returns a runtime error $\text{Err}(\text{RunTime})$.

Reification and reduction relation. The semantics for effects are given in terms of *reifiers*. A reifier for the signature E is a tuple (State, r) , where State is a type representing the internal state needed to reify the effects, and r is a reifier function of the type given in Figure 4. The idea is that r_i uses the internal state State to compute the output of the effect i based on its input.

For example, for the store effects we take *State* to be a map from locations to $\blacktriangleright \mathbf{IT}$ (representing the heap); and we define the following reifier functions:

$$\begin{aligned} r_{\text{write}}((\ell, \alpha), \sigma) &= \text{Some}((\ell, \sigma[\ell \mapsto \alpha])) && (\text{where } \ell \in \sigma, \text{ and None otherwise}) \\ r_{\text{read}}(\ell, \sigma) &= \text{Some}(\alpha, \sigma) && (\text{where } \sigma(\ell) = \alpha, \text{ and None otherwise}) \\ r_{\text{alloc}}(\alpha, \sigma) &= \text{Some}(\ell, \sigma[\ell \mapsto \alpha]) && (\text{where } \ell \notin \sigma) \end{aligned}$$

Given reifiers for all the effects, we define a function $\text{reify} : \mathbf{IT} \times \text{State} \rightarrow \mathbf{IT} \times \text{State}$ (as in Figure 4) that, given (α, σ) reifies the top-level effect in α using the state σ , and returns the reified GITree and the updated state.

The reify function is then used to give reduction semantics for GITrees. We write $(\alpha, \sigma) \rightsquigarrow (\beta, \sigma')$ for such a reduction step. The definition of $\rightsquigarrow$ is given internally in the logic:

$$\begin{aligned} (\alpha, \sigma) \rightsquigarrow (\beta, \sigma') &\triangleq (\alpha = \text{Tick}(\beta) \wedge \sigma = \sigma') \\ &\vee (\exists i x k. \alpha = \text{Vis}_i(x, k) \wedge \text{reify}(\alpha, \sigma) = (\text{Tick}(\beta), \sigma')) \end{aligned}$$

That is, either α is a “delayed” computation $\text{Tick}(\beta)$, which then reduces to β ; or it is an effect that can be reified. Recall that we write Tick for the composition $\text{Tau} \circ \text{next}$.

Note that the reify function operates on the top-level effect of the GITree. But what if the top-level constructor is not Vis , e.g. if we have an effect inside an “evaluation context”? The role of evaluation contexts in GITrees is played by *homomorphisms*, which also allow us to bubble up necessary effects to the top of the GITree.

Definition 1 (Homomorphism). A map $f : \mathbf{IT} \rightarrow \mathbf{IT}$ is a homomorphism, written $f \in \text{Hom}$, if it satisfies:

$$f(\text{Err}(e)) = \text{Err}(e) \quad f(\text{Tick}(\alpha)) = \text{Tick}(f(\alpha)) \quad f(\text{Vis}_i(x, k)) = \text{Vis}_i(x, \blacktriangleright f \circ k)$$

For example, $\lambda x. \alpha \bullet x$ is a homomorphism, and so is $\lambda x. \text{get\_val}(x, f)$. On the other hand, $\lambda x. \text{Vis}_{\text{alloc}}(\text{next}(x), k)$ (for some fixed k) is *not* a homomorphism.

Program logic. In order to reason about GITrees, we employ the full power of the Iris separation logic framework. The program logic operates on the propositions of the form $\text{wp } \alpha \{ \Phi \}$. This weakest precondition proposition intuitively states that the GITree α is safe to reduce, and when it fully reduces, the resulting value satisfies the predicate Φ . Another important predicate is $\text{has\_state}(\sigma)$, which signifies ownership of the current state σ .

In Figure 5 we show the rules, on which we focus in this work. Let us describe their meaning. The rule **WP-REIFY** allows us to symbolically execute effects in GITrees. It is given in a general form, and is used to derive domain-specific rules for concrete effects. Another important rule is **WP-HOM** which allows one to separate the reasoning about the computation from the reasoning about the context. The reason why **WP-HOM** is sound (this is going to be important in the next section when we make it unsound), is because the reduction $\rightsquigarrow$ of GITrees satisfies the following properties which allow one to disentangle a homomorphism from the GITree it’s applied to:

$$\begin{array}{c}
\text{WP-REIFY} \\
\frac{\text{has\_state}(\sigma) \quad \text{reify}(\text{Vis}_i(x, k), \sigma) = (\text{Tick}(\beta), \sigma') \quad \triangleright (\text{has\_state}(\sigma') \multimap \text{wp } \beta \{ \Phi \})}{\text{wp Vis}_i(x, k) \{ \Phi \}}
\end{array}
\qquad
\begin{array}{c}
\text{WP-HOM} \\
\frac{f \in \text{Hom} \quad \text{wp } \alpha \{ \beta_v. \text{wp } f(\beta_v) \{ \Phi \} \}}{\text{wp } f(\alpha) \{ \Phi \}}
\end{array}$$

Fig. 5. Selected weakest precondition rules.

Lemma 1. *Let f be a homomorphism. Then,*

- $(\alpha, \sigma) \rightsquigarrow (\beta, \sigma')$ implies $(f(\alpha), \sigma) \rightsquigarrow (f(\beta), \sigma')$;
- If $(f(\alpha), \sigma) \rightsquigarrow (\beta', \sigma')$ then either α is a *GITree*-value, or there exists β such that $(\alpha, \sigma) \rightsquigarrow (\beta, \sigma')$ and $\triangleright(f(\beta) = \beta')$.

Finally, as usual in Iris, the program logic satisfies an adequacy property, which allows one to relate propositions proved in the logic to the actual semantics:

Theorem 1. *Let α be an interaction tree and σ be a state such that*

$$\text{has\_state}(\sigma) \vdash \text{wp } \alpha \{ \Phi \}$$

is derivable for some meta-level predicate Φ (containing only intuitionistic logic connectives). Then for any β and σ' such that $(\alpha, \sigma) \rightsquigarrow^* (\beta, \sigma')$, one of the following two things hold:

- (adequacy) either $\beta \in \mathbf{IT}^v$, and $\Phi(\beta)$ holds in the meta-logic;
- (safety) or there are β_1 and σ_1 such that $(\beta, \sigma') \rightsquigarrow (\beta_1, \sigma_1)$

In particular, safety implies that $\beta \neq \text{Err}(e)$ for any error $e \in \mathbf{Error}$.

The role of meta-logic is played by the Coq system; thus, the adequacy theorem allows us to relate proofs inside the program logic (Iris) to the proofs on the level of Coq. This aspect is important in Iris and *GITrees* in general, but it is orthogonal to the work that we present in this paper. See [13] for more details.

3 Context-Dependent Reification

In this section we extend reification to handle context-dependent effects, using a language λ_{callcc} with `call/cc` as a concrete example. In Section 3.1 we present λ_{callcc} 's syntax and operational semantics (in the usual style with evaluation contexts). We then show why the current *GITrees* framework cannot be used as a denotational model for λ_{callcc} directly. In Section 3.2 we introduce our generalization of reification for context-dependent effects and corresponding extensions to the *GITrees* program logic. In Section 3.3 we demonstrate that our extension works as intended: we give a denotational semantics for λ_{callcc} , and we show how the general program logic for *GITrees* specializes to a logic for reasoning about `call/cc`. We prove soundness and computational adequacy of denotational semantics using a logical relation defined within our program logic.

types	$Ty \ni \tau ::= \mathbb{N} \mid \tau_1 \rightarrow \tau_2 \mid cont(\tau)$
expressions	$Expr \ni e ::= v \mid x \in Var \mid e_1 e_2 \mid e_1 \oplus e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ $\mid \text{call/cc } (x. e) \mid \text{throw } e_1 \text{ to } e_2$
values	$Val \ni v ::= n \mid \text{rec } f(x) = e \mid \text{cont } K$
eval. cont.	$Ectx \ni K ::= \square \mid \text{if } K \text{ then } e_1 \text{ else } e_2 \mid K v \mid e K \mid e \oplus K \mid K \oplus v$ $\mid \text{throw } K \text{ to } e \mid \text{throw } v \text{ to } K$
$\text{call/cc } (x. e) \mapsto_K e[\text{cont } K/x] \quad K[\text{throw } v \text{ to cont } K'] \mapsto K'[v] \quad \frac{e_1 \mapsto_K e_2}{K[e_1] \mapsto K[e_2]}$	
$\frac{\Gamma, x : cont(\tau) \vdash e : \tau}{\Gamma \vdash \text{call/cc } (x. e) : \tau} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : cont(\tau)}{\Gamma \vdash \text{throw } e_1 \text{ to } e_2 : \tau'}$	

Fig. 6. Syntax and fragments of type system and operational semantics of λ_{callcc} .

3.1 Operational Semantics and Type System for λ_{callcc}

By λ_{callcc} we denote a simply-typed λ -calculus with natural numbers, recursive functions, and **call/cc**. The relevant pieces of syntax, the type system and the operational semantics are given in Figure 6. The type system includes natural numbers, function types, and the type $cont(\tau)$ of continuations. The **call/cc** $(x. e)$ expression takes the current evaluation context and binds it to x in e . The **throw** e to e' expression evaluates passes the first argument (value) to the second argument (continuation, represented as an evaluation context).

The operational semantics of λ_{callcc} is separated into two layers. The first layer consists of local reductions of primitive expressions ($e \mapsto_K e'$), and the second layer lifts local reductions to reductions among complete programs ($K[e] \mapsto K'[e']$). The local reductions are parameterized by an evaluation context K , which allows **call/cc** $(x. e)$ to capture the evaluation context. Let us now consider what happens if we try to give a direct-style denotational semantics of λ_{callcc} into GITrees. By direct we mean that we wish to give a direct interpretation of types and expressions, rather than going through a global CPS conversion. To define the semantics, we first need to provide an effect signature, state, and reifiers for each effect, and then we can define the interpretation of the expressions of the language.

The effect signature, shown in Figure 7, contains two effects **callcc** and **throw**. Since **call/cc** binds a continuation, it is natural to let the input arity for **callcc** be a callback ($\blacktriangleright \mathbf{IT} \rightarrow \blacktriangleright \mathbf{IT}$) $\rightarrow \blacktriangleright \mathbf{IT}$. The output arity is simply $\blacktriangleright \mathbf{IT}$.

The input arity for **throw** signifies that **throw** takes as input an expression and a continuation, which are represented respectively as $\blacktriangleright \mathbf{IT}$ and $\blacktriangleright (\mathbf{IT} \rightarrow \mathbf{IT})$. The output arity of **throw** is simply the empty type 0, because **throw** never returns.

Note that the input types of **callcc** and **throw** have slightly different arities. However, we can always transform $f : (\blacktriangleright X \rightarrow \blacktriangleright X)$ into an element of type $\blacktriangleright (X \rightarrow X)$ by performing a silent step in the function's body: $f' \triangleq \text{next}(\lambda x. \text{Tau}(f(\text{next}(x))))$. And we can always transform $f : \blacktriangleright (X \rightarrow X)$ into an element of type $\blacktriangleright X \rightarrow \blacktriangleright X$ using the applicative structure of the later modality.

$$\begin{aligned}
Ins_{\text{callcc}}(X) &\triangleq ((\blacktriangleright X \rightarrow \blacktriangleright X) \rightarrow \blacktriangleright X) & Outs_{\text{callcc}}(X) &\triangleq \blacktriangleright X \\
Ins_{\text{throw}}(X) &\triangleq \blacktriangleright X \times \blacktriangleright (X \rightarrow X) & Outs_{\text{throw}}(X) &\triangleq 0 \\
\text{Callcc}(f) &\triangleq \text{Vis}_{\text{callcc}}(f, \text{id}) & \text{Throw}(e, f) &\triangleq \text{Vis}_{\text{throw}}(e, f, \lambda x. \text{abort } x)
\end{aligned}$$

Fig. 7. Signatures and operations on GITrees with `call/cc`.

$$\begin{aligned}
r : \prod_{i \in E} Ins_i(\mathbf{IT}_E) \times State \times (Outs_i(\mathbf{IT}_E) \rightarrow \blacktriangleright \mathbf{IT}_E) &\rightarrow option(\blacktriangleright \mathbf{IT}_E \times State) \\
\frac{r_i(x, \sigma, \kappa) = \text{Some}(\beta, \sigma')}{\text{reify}(\text{Vis}_i(x, \kappa), \sigma) = (\text{Tau}(\beta), \sigma')} & \qquad \frac{r_i(x, \sigma, \kappa) = \text{None}}{\text{reify}(\text{Vis}_i(x, \kappa), \sigma) = (\text{Err}(\text{RunTime}), \sigma)}
\end{aligned}$$

Fig. 8. Type of context-dependent reifiers and the context-dependent reify function

For convenience, we will use the abbreviations $\text{Callcc}(f)$ and $\text{Throw}(e)$, defined in Figure 7, for representing denotations of `throw` and `call/cc` as effects in GITrees.

To complete all the ingredients for the denotational semantics, we need reifiers for the `callcc` and `throw` effects. Given our operational understanding of continuations, the natural choice for the local state type *State* is **1** (since we do not have any state). However, the current reifier signature (Figure 4) poses a problem. Reifiers, as they are now, cannot access their current continuation, which is essential for both effects. $\text{Callcc}(f)$ needs to pass the current continuation to f , while Throw must redirect control to a provided continuation instead of returning normally. The current reifiers lacks this capability, and in the next subsection we show how to generalize the notion of reification to context-dependent effects.

3.2 Context-dependent Reifiers

This section presents our extension to context-dependent reification, and the limitations it imposes on the program logic. In order to allow reifiers to manage continuations, we change the type of reifiers to accept continuations as an extra parameter, as shown in Figure 8. Continuations for a given effect are functions from the effect's outputs to GITrees: $Outs_i(\mathbf{IT}_E) \rightarrow \blacktriangleright \mathbf{IT}_E$. Given a set of context-dependent reifiers, we define a context-dependent **reify** function, also shown in Figure 8. As before, **reify** dispatches to the correct individual reifier for the effect. Note that now it is the user's responsibility to pass the output of an effect to the given continuation if the control flow is not supposed to be interrupted. For example, since the evaluation of a `call/cc` ($x. e$) expression does not modify the control flow itself, but simply passes the current continuation to its body, the context-dependent reifier for `callcc` is simply $r_{\text{callcc}}(x, \sigma, \kappa) = \text{Some}(\kappa(x \ \kappa), \sigma)$.

Before we move on to discussing the consequences of this for program logic, we would like to note that our treatment of continuation (with top-level reifiers

WP-WRITE	WP-WRITE-CTX-DEP	WP-REIFY-CTX-DEP
$\kappa \in Hom$		$has\_state(\sigma)$
$heap\_ctx \triangleright \ell \mapsto \alpha$	$heap\_ctx \triangleright \ell \mapsto \alpha$	$r_i(x, \sigma, k) = Some(next(\beta), \sigma')$
$\triangleright (\ell \mapsto \beta \multimap)$	$\triangleright (\ell \mapsto \beta \multimap)$	$\triangleright (has\_state(\sigma') \multimap)$
$wp\ Ret() \{ \Phi \}$	$wp\ \kappa\ (Ret()) \{ \Phi \}$	$wp\ \beta \{ \Phi \}$
$wp\ Write(\ell, \beta) \{ \Phi \}$	$wp\ \kappa\ (Write(\ell, \beta)) \{ \Phi \}$	$wp\ Vis_i(x, k) \{ \Phi \}$

Fig. 9. Program logic in the presence of context-dependent reifiers.

dispatching them) parallels Cartwright and Felleisen’s “extensible direct models” [9], which also aimed to support extensible denotational semantics in classical domain theory. We discuss this more in [Section 6](#).

Program logic for GITrees in the presence of context-dependent reifiers. To reflect the generalization to context-dependent reifiers in the program logic, we replace the proof rule [WP-REIFY](#) by [WP-REIFY-CTX-DEP](#), shown in [Figure 9](#). This is, however, not the only change we need to make. In the presence of context-dependent effects, [WP-HOM](#) is not sound! (A similar observation was also made by [28] in their development of a program logic for `call/cc`.) The reason is that context-dependent reification invalidates [Lemma 1](#). Now, since [WP-HOM](#) is not sound anymore, one might expect that we need to adapt all the other program logic rules to include a homomorphism similarly to how the rules of [28] were adapted to include an evaluation context. However, this is not necessary, because our program logic is defined on *denotations* on which we have a non-trivial equational theory, which can be used to reason about ‘pure’ GITrees. Only for effectful operations, the proof rules will now have to include a surrounding homomorphism. E.g., [WP-WRITE](#) from [13] is generalized to [WP-WRITE-CTX-DEP](#), and considers ambient homomorphisms explicitly.

Our context-dependent reification extension, though simple, allows us to build sound and adequate denotational models for languages with control-flow operators, including λ_{callcc} (shown in the next subsection). Moreover, our extension is conservative, and we recover previous case studies (computational adequacy of $\lambda_{rec,io}$ and type safety for $\lambda_{\multimap,ref}$ [13]) with minimal modifications; see the accompanying Coq formalization.

3.3 Denotational Semantics of λ_{callcc}

In this section we show that context-dependent reifiers are sufficient for providing a sound and adequate semantic model of λ_{callcc} . We define context-dependent reifiers for `callcc` and `throw`, then prove that this gives a sound interpretation w.r.t. operational semantics. To show adequacy, we define a logical relation, which relates the denotational and operational semantics. The logical relation is defined in the (updated) program logic for GITrees (following the approach in [13]), and validates the utility of the program logic.

$$\begin{aligned}
\mathbf{E}\llbracket x \rrbracket_\rho &= \rho(x) \\
\mathbf{E}\llbracket \text{call/cc } (\lambda x. e) \rrbracket_\rho &= \text{Callcc}(\lambda(f : \blacktriangleright \mathbf{IT} \rightarrow \blacktriangleright \mathbf{IT}). \mathbf{E}\llbracket e \rrbracket_{\rho[x \mapsto \text{Fun}(\text{next}(\lambda y. \text{Tau}(f(\text{next}(y))))])}) \\
\mathbf{E}\llbracket \text{throw } e_1 \text{ to } e_2 \rrbracket_\rho &= \text{get\_val}(\mathbf{E}\llbracket e_1 \rrbracket_\rho, \lambda x. \text{get\_fun}(\mathbf{E}\llbracket e_2 \rrbracket_\rho, \lambda f. \text{Throw}(x, f))) \\
\mathbf{V}\llbracket \text{cont } K \rrbracket_\rho &= \text{Fun}(\text{next}(\lambda x. \text{Tau}(\mathbf{K}\llbracket K \rrbracket_\rho (\blacktriangleright \bullet) \text{next}(x)))) \\
\mathbf{K}\llbracket \text{throw } K \text{ to } e \rrbracket_\rho &= \lambda x. \text{get\_val}(\mathbf{K}\llbracket K \rrbracket_\rho x, \lambda y. \text{get\_fun}(\mathbf{E}\llbracket e \rrbracket_\rho, \lambda f. \text{Throw}(y, f))) \\
\mathbf{K}\llbracket \text{throw } v \text{ to } K \rrbracket_\rho &= \lambda x. \text{get\_val}(\mathbf{V}\llbracket v \rrbracket_\rho, \lambda y. \text{get\_fun}(\mathbf{K}\llbracket K \rrbracket_\rho x, \lambda f. \text{Throw}(y, f)))
\end{aligned}$$

Fig. 10. Denotational semantics of λ_{callcc} (selected clauses).

Interpretation of λ_{callcc} . The denotational semantics of λ_{callcc} is shown in [Figure 10](#) (selected clauses only; see Coq formalization for the complete definition). The interpretation is split into three parts: $\mathbf{E}\llbracket - \rrbracket$ for expressions, $\mathbf{V}\llbracket - \rrbracket$ for values, and $\mathbf{K}\llbracket - \rrbracket$ for contexts. For the interpretation of `throw  $e_1$  to  $e_2$` , the left-to-right evaluation order is enforced by the functions `get_val` and `get_fun`. They first evaluate their argument to a `GITree` value, and then pass it on (c.f. [Figure 3](#)).

The context-dependent reifiers for the effects `callcc` and `throw` are defined as follows:

$$r_{\text{callcc}}(f, (), \kappa) = \text{Some}(\kappa (f \kappa), ()) \quad r_{\text{throw}}((\alpha, f), (), \kappa) = \text{Some}(f \alpha, ())$$

To show that the denotational semantics is sound, we need the following lemma that shows that interpretations of expressions in evaluation contexts are decomposed into applications of homomorphisms.

Lemma 2. *For any context K and an environment ρ , we have $\mathbf{K}\llbracket K \rrbracket_\rho \in \text{Hom}$. For any context K , expression e , and an environment ρ , $\mathbf{E}\llbracket K[e] \rrbracket_\rho = \mathbf{K}\llbracket K \rrbracket_\rho(\mathbf{E}\llbracket e \rrbracket_\rho)$.*

With these results at hand, we can show soundness of our interpretation:

Lemma 3. *Soundness. Suppose $e_1 \mapsto e_2$. Then $(\mathbf{E}\llbracket e_1 \rrbracket_\rho, ()) \rightsquigarrow^* (\mathbf{E}\llbracket e_2 \rrbracket_\rho, ())$, where $() : \mathbf{1}$ is the unique element of the unit type, representing the (lack of) state.*

Program logic for λ_{callcc} . We now specialize the general program logic rule [WP-REIFY-CTX-DEP](#) using the reifiers for `callcc` and `throw` to obtain the following program logic rules:

$$\begin{array}{c}
\text{WP-THROW} \\
\frac{\kappa \in \text{Hom} \quad \text{has\_state}(\sigma) \quad \triangleright(\text{has\_state}(\sigma) * \text{wp } f \ x \ \{\Phi\})}{\text{wp } \kappa \ (\text{Throw}(\text{next}(x), \text{next}(f))) \ \{\Phi\}}
\end{array}
\qquad
\begin{array}{c}
\text{WP-CALLCC} \\
\frac{\kappa \in \text{Hom} \quad \text{has\_state}(\sigma) \quad \triangleright(\text{has\_state}(\sigma) * \text{wp } \kappa \ (f \ \kappa) \ \{\Phi\})}{\text{wp } \kappa \ (\text{Callcc}(\text{next} \circ f)) \ \{\Phi\}}
\end{array}$$

where κ is a homomorphism representing the current evaluation context on the level of `GITrees`. The reader may wonder why these rules include the `has_state`(σ) predicates, since it is just ‘threaded around’. The reason is that these rules also apply when there are other effects around and the state is composed of different substates for different effects, cf. the discussion of modularity in [Section 2](#).

$$\begin{aligned}
\mathcal{O}(\alpha, e) &\triangleq \text{has\_state}() \multimap \text{wp } \alpha \{ \beta. \exists v. (e \mapsto^* v) * \llbracket \mathbb{N} \rrbracket(\beta, v) * \text{has\_state}() \} \\
&\quad \boxed{\mathcal{O} : \text{ERel}} \\
\mathcal{K}(R)(\kappa, K) &\triangleq \square \forall (\beta, v). R(\beta, v) \multimap \mathcal{O}(\kappa \beta, K[v]) \\
&\quad \boxed{\mathcal{K} : \text{VRel} \rightarrow \text{CRel}} \\
\mathcal{E}(R)(\alpha, e) &\triangleq \forall (\kappa, K). \mathcal{K}(R)(\kappa, K) \multimap \mathcal{O}(\kappa \alpha, K[e]) \\
&\quad \boxed{\mathcal{E} : \text{VRel} \rightarrow \text{ERel}} \\
\llbracket \mathbb{N} \rrbracket(\alpha, v) &\triangleq \exists n : \mathbb{N}. \alpha = \text{Ret}(n) \wedge v = n \\
&\quad \boxed{\llbracket \tau \rrbracket : \text{VRel}} \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket(\beta, v) &\triangleq \exists f. \beta = \text{Fun}(f) \wedge \square \forall (\alpha, v'). \llbracket \tau_1 \rrbracket(\alpha, v') \multimap \mathcal{E}(\llbracket \tau_2 \rrbracket)(\text{Fun}(f) \bullet \alpha, v v') \\
\llbracket \text{cont}(\tau) \rrbracket(\beta, v) &\triangleq \exists \kappa. K. \beta = \text{Fun}(\text{next}(\lambda x. \text{Tick}(\kappa x))) \wedge v = \text{cont } K \wedge \mathcal{K}(\llbracket \tau \rrbracket)(\kappa, K) \\
\llbracket \Gamma \rrbracket(\rho, \gamma) &\triangleq \forall (x : \tau) \in \Gamma. \llbracket \tau \rrbracket(\rho x, \gamma x) \\
\Gamma \vdash e : \tau &\triangleq \square \forall (\rho, \gamma). \llbracket \Gamma \rrbracket(\rho, \gamma) \multimap \mathcal{E}(\llbracket \tau \rrbracket)(\mathbf{E}[e]_\rho, e[\gamma]) \\
&\quad \boxed{\begin{aligned} \text{CRel} &\triangleq \text{Hom} \times \text{Ectx} \rightarrow i\text{Prop} \\ \text{VRel} &\triangleq \mathbf{IT}^v \times \text{Val} \rightarrow i\text{Prop} \\ \text{ERel} &\triangleq \mathbf{IT} \times \text{Expr} \rightarrow i\text{Prop} \end{aligned}}
\end{aligned}$$

Fig. 11. Logical relation for λ_{callcc} .

Adequacy and logical relation. Having established soundness, we now turn our attention to *adequacy*, which is usually much more complicated to prove.

Lemma 4. *Adequacy.* Suppose that $\emptyset \vdash e : \mathbb{N}$ and $(\mathbf{E}[e]_\emptyset, \sigma_1) \rightsquigarrow^* (\text{Ret}(n), \sigma_2)$, for a natural number n . Then $e \mapsto^* n$.

To prove [Lemma 4](#), we define a logical relation between syntax (λ_{callcc} programs) and semantics (GITrees denotations) using the program logic from [Figure 11](#). To handle control effects, we use a *biorthogonal* logical relation [23], adapted from [28] for adequacy, following the Iris approach [30].

The core observational refinement $\mathcal{O}(\alpha, e)$ ensures that if α reduces to a GITree value $\mathbf{IT}^v$, then this value is a natural number, and e also reduces to the same number. The evaluation context relation $\mathcal{K}(P)(\kappa, K)$ relates homomorphisms and evaluation contexts when they map related arguments to expressions satisfying $\mathcal{O}$. The expression relation $\mathcal{E}(P)(\alpha, e)$ connects related $\mathbf{IT}$'s and expressions in related evaluation contexts. Types are inductively interpreted: functions relate if they map related arguments to related results, and continuations relate via the context relation. For open terms, the validity judgment $\Gamma \vdash e : \tau$ uses closing substitutions, with $e[\gamma]$ denoting applying a substitution γ to e .

The proof of adequacy relies on the fact that the interpretation of evaluation contexts are homomorphisms, which allows us to use a limited version of the bind rule:

Lemma 5. *Limited bind rule.* If $\mathcal{E}(P)(\alpha, e)$ and $\mathcal{K}(P)(\kappa, K)$, then $\mathcal{O}(\kappa \alpha, K[e])$.

With this in mind we show the fundamental lemma, stating that every well-typed expression is related to its own interpretation:

Lemma 6. *Fundamental lemma.* Let $\Gamma \vdash e : \tau$ then $\Gamma \vdash e : \tau$.

types	$Ty \ni \tau, \sigma, \alpha, \beta, \delta, \gamma ::= \mathbb{N} \mid \tau/\alpha \rightarrow \sigma/\beta \mid cont(\tau, \alpha)$	$\Gamma; \alpha \vdash e : \tau; \beta$ $\Gamma \vdash_{\text{pure}} e : \tau$		
expressions	$Expr \ni e ::= v \mid x \mid e_1 e_2 \mid e_1 \oplus e_2$ $\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \mathcal{S} x. e \mid \mathcal{D} e \mid e_1 @ e_2$			
values	$Val \ni v ::= n \mid \text{rec } f(x) = e \mid \text{cont } K$			
eval. contexts	$Ectx \ni K ::= \square \mid K[\text{if } \square \text{ then } e_1 \text{ else } e_2] \mid K[v \square] \mid K[\square e]$ $\mid K[e \oplus \square] \mid K[\square \oplus v] \mid K[\square @ v] \mid K[e @ \square]$			
	$\frac{\Gamma \vdash_{\text{pure}} e : \tau}{\Gamma; \alpha \vdash e : \tau; \alpha}$	$\frac{\Gamma, x : \text{cont}(\tau, \alpha); \sigma \vdash e : \sigma; \beta}{\Gamma; \alpha \vdash \mathcal{S} x. e : \tau; \beta}$	$\frac{\Gamma; \tau \vdash e : \tau; \sigma}{\Gamma \vdash_{\text{pure}} \mathcal{D} e : \sigma}$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{pure}} x : \tau}$
	$\frac{\Gamma, f : \sigma/\alpha \rightarrow \tau/\beta, x : \sigma; \alpha \vdash e : \tau; \beta}{\Gamma \vdash_{\text{pure}} \text{rec } f(x) = e : \sigma/\alpha \rightarrow \tau/\beta}$		$\frac{\Gamma; \gamma \vdash e_1 : \sigma/\alpha \rightarrow \tau/\beta; \delta \quad \Gamma; \beta \vdash e_2 : \sigma; \gamma}{\Gamma; \alpha \vdash e_1 e_2 : \tau; \delta}$	
	$\frac{\Gamma; \beta \vdash e_1 : \mathbb{N}; \alpha \quad \Gamma; \sigma \vdash e_2 : \tau; \beta \quad \Gamma; \sigma \vdash e_3 : \tau; \beta}{\Gamma; \sigma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau; \alpha}$		$\overline{\Gamma \vdash_{\text{pure}} n : \mathbb{N}}$	
	$\frac{\Gamma; \alpha \vdash e_1 : \mathbb{N}; \beta \quad \Gamma; \beta \vdash e_2 : \mathbb{N}; \sigma}{\Gamma; \alpha \vdash e_1 \oplus e_2 : \mathbb{N}; \sigma}$	$\frac{\Gamma; \sigma \vdash e_1 : \text{cont}(\tau, \alpha); \delta \quad \Gamma; \delta \vdash e_2 : \tau; \beta}{\Gamma; \sigma \vdash e_1 @ e_2 : \alpha; \beta}$		

Fig. 12. Syntax and typing rules of λ_{delim} .

Computational adequacy now follows easily from the fundamental lemma.

Proof (of Lemma 4). By Lemma 6, we have that $\emptyset \vdash e : \mathbb{N}$ implies that $\emptyset \models e : \mathbb{N}$. Now, the statement follows from Theorem 1 and the assumption that $\mathbf{E}[e]_{\emptyset}, \sigma_1 \rightsquigarrow^* \text{Ret } n, \sigma_2$.

4 Modeling Delimited Continuations

In this section we scale our approach to delimited continuations, which is a challenging example of context-dependent effects. We provide a denotational semantics for a programming language λ_{delim} with **shift/reset**, and prove its soundness and adequacy relative to an abstract machine semantics [6]. The semantics and proofs are more complex than for λ_{callcc} due to the nature of delimited continuations and associated type system. To the best of our knowledge, this represents the first formalized sound and adequate direct-style denotational semantics for delimited continuations.

4.1 Syntax and Operational Semantics of λ_{delim}

The syntax and the type system of λ_{delim} is given in Figure 12. It is similar to λ_{callcc} , but instead of **call/cc** ($-.$ $-$) there are operators $\mathcal{D} e$ (delimit the current evaluation context, also known as **reset**) and $\mathcal{S} x. e$ (grab the current delimited continuation, and bind it to x in e , also known as **shift**).

$\langle e \rangle_{\text{term}} \mapsto \langle e, \square, [] \rangle_{\text{eval}}$	metacontinuations:
$\langle K :: mk, v \rangle_{\text{mcont}} \mapsto \langle K, v, mk \rangle_{\text{cont}}$	$M\text{cont} \ni mk ::= [] \mid K :: mk$
$\langle [], v \rangle_{\text{mcont}} \mapsto \langle v \rangle_{\text{ret}}$	abstract machine config.:
$\langle \square, v, mk \rangle_{\text{cont}} \mapsto \langle mk, v \rangle_{\text{mcont}}$	$\text{Config} ::= \langle e, K, mk \rangle_{\text{eval}} \mid$
$\langle K[\square @ v], \text{cont } K', mk \rangle_{\text{cont}} \mapsto \langle K', v, K :: mk \rangle_{\text{cont}}$	$\langle K, v, mk \rangle_{\text{cont}} \mid$
$\langle K[e @ \square], v, mk \rangle_{\text{cont}} \mapsto \langle e, K[\square @ v], mk \rangle_{\text{eval}}$	$\langle mk, v \rangle_{\text{mcont}} \mid$
$\langle v, K, mk \rangle_{\text{eval}} \mapsto \langle K, v, mk \rangle_{\text{cont}}$	$\langle e \rangle_{\text{term}} \mid$
$\langle e_0 @ e_1, K, mk \rangle_{\text{eval}} \mapsto \langle e_1, K[e_0 @ \square], mk \rangle_{\text{eval}}$	$\langle v \rangle_{\text{ret}}$
$\langle \mathcal{D} \ e, K, mk \rangle_{\text{eval}} \mapsto \langle e, \square, K :: mk \rangle_{\text{eval}}$	
$\langle \mathcal{S} \ k. e, K, mk \rangle_{\text{eval}} \mapsto \langle e[K/k], \square, mk \rangle_{\text{eval}}$	

Fig. 13. Operational semantics of λ_{delim} (excerpt).

The type system follows Danvy and Filinski [10], extending simply-typed λ -calculus with answer types α, β . The main typing judgment $\Gamma; \alpha \vdash e : \tau; \beta$ means: under the typing context Γ , expression e can be plugged into a context expecting a value of type τ and producing a value of type α ; in that case the resulting program will have the type β . You can think of it a computation $\Gamma \rightarrow (\tau \rightarrow \alpha) \rightarrow \beta$ under the CPS translation. Thus, the type of the (delimited) continuation corresponds to $\tau \rightarrow \alpha$, while the type of the overall expression is β . The pure typing judgment $\Gamma \vdash_{\text{pure}} e : \tau$ indicates e does not depend on the surrounded context, and is context-independent for any answer types. Expressions can change their context's answer type, as seen in the $\mathcal{S} \ x. e$ typing rule.

For example, suppose we extend the type system with booleans, $\mathbb{B}$, and add a primitive function `isprime` that does not modify answer types. That is $\emptyset; \alpha \vdash \text{isprime} : \mathbb{N}/\beta \rightarrow \mathbb{B}/\beta; \alpha$. The expression $\mathcal{D} \ ((\text{recf}(x) = \text{isprime} \ (\mathcal{S} \ k. x - 1)) \ 2)$ is well-typed in this type system as a $\mathbb{N}$, even though it changes the answer type from $\mathbb{B}$ to $\mathbb{N}$.

Answer types appear in both judgments and type constructors. Continuation type `cont` (τ, α) represents contexts expecting something of the type τ and producing something of the type α . Function type $\sigma/\alpha \rightarrow \tau/\beta$, in addition to the input type σ and the output type τ , record the typing of the surrounding context at the point of the function call. See [10] for details.

The operational semantics for λ_{delim} uses a CEK machine, following [6, 12, 5]. Selected reduction rules appear in Figure 13 (see Coq formalization for the full set of rules). The abstract machine operates on various *configurations*, which can be of several forms. The first one is the initial configuration $\langle e \rangle_{\text{term}}$, which is just a starting state for evaluating expressions. Similarly, there is a terminal configuration $\langle v \rangle_{\text{ret}}$ signifying that the program has terminated with the value v .

From the initial configuration, we go on to $\langle e, K, mk \rangle_{\text{eval}}$, which signifies that we are evaluating an expression e inside the current delimited context K , with the metacontinuation mk (a stack of continuations based on different delimiters). It

$$\begin{array}{ll}
\text{Ins}_{\text{reset}}(X) \triangleq \blacktriangleright X & \text{Ins}_{\text{shift}}(X) \triangleq (\blacktriangleright X \rightarrow \blacktriangleright X) \rightarrow \blacktriangleright X \\
\text{Ins}_{\text{pop}}(X) \triangleq \blacktriangleright X & \text{Ins}_{\text{appcont}}(X) \triangleq \blacktriangleright X \times \blacktriangleright (X \rightarrow X) \\
\text{Outs}_{\text{reset}}(X) \triangleq \blacktriangleright X & \text{Outs}_{\text{shift}}(X) \triangleq \blacktriangleright X \\
\text{Outs}_{\text{pop}}(X) \triangleq 0 & \text{Outs}_{\text{appcont}}(X) \triangleq \blacktriangleright X \\
r_{\text{reset}}(e, \sigma, \kappa) = \text{Some}(e, \kappa :: \sigma) & r_{\text{shift}}(f, \sigma, \kappa) = \text{Some}(f \ \kappa, \sigma) \\
r_{\text{pop}}(e, [], \_) = \text{Some}(e, []) & r_{\text{pop}}(e, \kappa :: \sigma, \_) = \text{Some}(\kappa \ e, \sigma) \\
r_{\text{appcont}}((e, \kappa), \sigma, \kappa') = \text{Some}(\kappa \ e, \kappa' :: \sigma) & \mathbf{P}(\beta) \triangleq \text{get\_val}(\beta, \text{Pop}) \\
\text{Reset}(e) \triangleq \text{Vis}_{\text{reset}}(e, \text{id}) & \text{Shift}(f) \triangleq \text{Vis}_{\text{shift}}(f, \text{id}) \\
\text{Appcont}(e, f) \triangleq \text{Vis}_{\text{appcont}}((e, f), \text{id}) & \text{Pop}(e) \triangleq \text{Vis}_{\text{pop}}(e, \lambda x. \text{abort } x)
\end{array}$$

Fig. 14. Effects for λ_{delim} .

is this configuration type which takes care of delimited control operations. The $\mathcal{D}$ operator saves the current continuation on top of the metacontinuation, limiting the scope of $\mathcal{S} \ x. e$. The $\mathcal{S} \ x. e$ operation behaves similarly to `call/cc` ($x. e$), except that it prevents later control operators from capturing its evaluation context.

The last two configuration types are for dealing with continuations and metacontinuations. A configuration $\langle K, v, mk \rangle_{\text{cont}}$ signifies that we are trying to plug in the value v into the context K , with the metacontinuation mk . A configuration $\langle mk, v \rangle_{\text{mcont}}$ signifies that we are done with the current continuation (ending with the value v), but we still have to unwind the continuation stack mk .

4.2 Denotational Semantics of λ_{delim}

Our model represents delimited continuations with effects mimicking an abstract machine, operating on semantic rather than syntactic components. The effect signature and reifiers (Figure 14) define a state with a stack of continuations, manipulated explicitly. The effect signature $E_{\lambda_{\text{delim}}}$ includes four operators: `{reset, shift, pop, appcont}`. The signature of `reset` simply tells us that the corresponding effect does not directly modify its argument. The auxiliary effect `pop`, which does not have an equivalent in the surface syntax, is used to enforce unwinding of the continuation stack. As the output arity of `pop` signifies, it does not return. We describe the importance of that below. The rest of the signature is more straightforward: `shift` and `appcont` are defined exactly as `callcc` and `throw`. The semantics of these effects, in terms of reification, is more intricate. As we mentioned, the state for reification is $\text{State} = \text{List}(\blacktriangleright \mathbf{IT} \rightarrow \blacktriangleright \mathbf{IT})$.

In comparison with `call/cc` ($x. e$), the control operator $\mathcal{S} \ x. e$ does not necessarily continue from the same continuation; hence, the corresponding reifier passes the current continuation to the body, but does not return control back. The reifier for `reset` simply saves the current continuation κ onto the stack σ . It is then the job of the `pop` operation to restore the continuation from the stack.

$$\begin{array}{c}
\text{WP-SHIFT} \\
\frac{\text{has\_state}(\sigma) \quad \triangleright(\text{has\_state}(\sigma) \multimap \text{wp } \beta \{ \Phi \}) \quad \blacktriangleright \mathbf{P}(f(\blacktriangleright \kappa)) = \text{next}(\beta)}{\text{wp } \kappa(\text{Shift}(f)) \{ \Phi \}}
\end{array}
\qquad
\begin{array}{c}
\text{WP-RESET} \\
\frac{\text{has\_state}(\sigma) \quad \triangleright(\text{has\_state}(\blacktriangleright \kappa :: \sigma) \multimap \text{wp } \mathbf{P}(e) \{ \Phi \})}{\text{wp } \kappa(\text{Reset}(\text{next}(e))) \{ \Phi \}}
\end{array}$$

$$\begin{array}{c}
\text{WP-POP} \\
\frac{\text{has\_state}(\sigma) \quad \kappa' = \kappa \text{ if } \sigma = \kappa :: \sigma' \text{ and id otherwise} \quad \triangleright(\text{has\_state}(\text{tail}(\sigma)) \multimap \text{wp } \kappa'(v) \{ \Phi \})}{\text{wp } \mathbf{P}(v) \{ \Phi \}}
\end{array}
\qquad
\begin{array}{c}
\text{WP-APPCONT} \\
\frac{\text{has\_state}(\sigma) \quad \triangleright(\text{has\_state}(\blacktriangleright \kappa :: \sigma) \multimap \text{wp } \beta \{ \Phi \}) \quad \blacktriangleright \kappa'(e) = \text{next}(\beta)}{\text{wp } \kappa(\text{Appcont}(e, \kappa')) \{ \Phi \}}
\end{array}$$

Fig. 15. Weakest precondition rules for delimited continuations.

$$\begin{aligned}
\mathbf{E}[\![\mathcal{D} \ e]\!]_{\rho} &= \text{Reset}(\mathbf{P}(\mathbf{E}[\![e]\!]_{\rho})) \\
\mathbf{E}[\![\mathcal{S} \ x. \ e]\!]_{\rho} &= \text{Shift}(\mathbf{P} \circ (\lambda \kappa. \mathbf{E}[\![e]\!]_{\rho, x \mapsto \text{Fun}(\text{next}(\lambda y. \text{Tau}(\kappa(\text{next}y))))})) \\
\mathbf{E}[\![e_1 \ @ \ e_2]\!]_{\rho} &= \text{get\_val}(\mathbf{E}[\![e_2]\!]_{\rho}, \lambda x. \text{get\_fun}(\mathbf{E}[\![e_1]\!]_{\rho}, \lambda y. \text{Appcont}(\text{next}(x), y))) \\
\mathbf{V}[\![\text{cont } K]\!]_{\rho} &= \text{Fun}(\text{next}(\lambda x. \text{Tick}(\mathbf{P}(\mathbf{K}[\![K]\!]_{\rho} \ x)))) \\
\mathbf{K}[\![K[\Box \ @ \ v]]\!]_{\rho} &= \lambda x. \mathbf{K}[\![K]\!]_{\rho}(\mathbf{E}[\![x \ @ \ v]\!]_{\rho}) \\
\mathbf{M}[\![mk]\!]_{\rho} &= \text{map}(\lambda k. \mathbf{P} \circ \mathbf{K}[\![k]\!]_{\rho})mk \\
\mathbf{S}[\![\langle e, K, mk \rangle_{\text{eval}}]\!]_{\rho} &= (\mathbf{P}(\mathbf{E}[\![K[e]\!]_{\rho}]), \mathbf{M}[\![mk]\!]_{\rho}) \\
\mathbf{S}[\![\langle K, v, mk \rangle_{\text{cont}}]\!]_{\rho} &= (\mathbf{P}(\mathbf{E}[\![K[v]\!]_{\rho}]), \mathbf{M}[\![mk]\!]_{\rho}) \\
\mathbf{S}[\![\langle mk, v \rangle_{\text{mcont}}]\!]_{\rho} &= (\mathbf{P}(\mathbf{V}[\![v]\!]_{\rho}), \mathbf{M}[\![mk]\!]_{\rho}) \\
\mathbf{S}[\![\langle e \rangle_{\text{term}}]\!]_{\rho} &= (\mathbf{P}(\mathbf{E}[\![e]\!]_{\rho}), []) \\
\mathbf{S}[\![\langle v \rangle_{\text{ret}}]\!]_{\rho} &= (\mathbf{V}[\![v]\!]_{\rho}, [])
\end{aligned}$$

Fig. 16. Denotational semantics for a calculus with delimited control (selected clauses).

The reifier for `shift` is similar to that of `callcc`, except that it removes the current continuation entirely. The reifier for `appcont`, in comparison with `throw`, does not simply pass control, but also saves the current continuation on the stack. This corresponds to the fact that whenever a delimited continuation is invoked, the result is wrapped in a `reset`; that is done to prevent the continuation from escaping the delimiter. As part of instantiating `GITrees` with these effects, we obtain the specialized program logic rules shown in Figure 15. We will use those rules later for defining a logical relation between the syntax and the semantics of λ_{delim} . As mentioned above, we will use `Pop` to unwind the continuation stack and restore the continuation after finishing with a `reset`. This means that we will need to insert explicit calls to `Pop` in the interpretation of λ_{delim} . For these purposes, we use an abbreviation $\mathbf{P}(\beta)$, which first evaluates β to a value, and then executes the `pop` operation.

The interpretation of λ_{delim} uses this auxiliary function and is given in [Figure 16](#). Similarly to the operational semantics, the interpretation is divided into five categories. First, we have $\mathbf{E}[-]$ and $\mathbf{V}[-]$ for the interpretation of expressions and values, which is what we need for the surface syntax. All of those interpretations return GITrees. Note that in the interpretation of $\mathcal{D}$ – we insert explicit calls to $\mathbf{P}$, and similarly in the interpretation of continuations.

The other group of interpretations, $\mathbf{K}[-]$, $\mathbf{M}[-]$ and $\mathbf{S}[-]$, are for interpreting continuations, metacontinuations, and other configurations; these are used for showing soundness (preservation of operational semantics by the interpretation). The interpretation $\mathbf{K}[-]$ of continuations returns a semantic continuation (a function $\mathbf{IT} \rightarrow \mathbf{IT}$). Similarly, the interpretations $\mathbf{M}[-]$ (resp. $\mathbf{S}[-]$) of metacontinuations (resp. configurations) returns a stack of semantic continuations (resp. a semantic configuration).

We now show that our interpretation is sound w.r.t. the abstract machine semantics. For this we prove lemmas similar to [Lemma 2](#), and put them to use in the soundness theorem:

Theorem 2. *Soundness. Let $c_0, c_1 \in \text{Config}$ and suppose $c_0 \mapsto c_1$. Then $\mathbf{S}[c_0] \rightsquigarrow^* \mathbf{S}[c_1]$.*

4.3 Logical Relation and Adequacy

We now show that our denotational semantics is adequate with regards to the abstract machine semantics. Specifically, we show the following result:

Theorem 3. *Adequacy. Suppose $\emptyset; \mathbb{N} \vdash e : \mathbb{N}; \mathbb{N}$ is a well-typed term, and that $(\mathbf{P}(\mathbf{E}[e]_\emptyset), []) \rightsquigarrow^* (\text{Ret}(n), \sigma)$ for a natural number n and a metacontinuation σ . Then $\langle e \rangle_{\text{term}} \mapsto^* \langle n \rangle_{\text{ret}}$.*

We prove adequacy using a logical relation. It relates expressions to their interpretations and also connects syntactic and semantic configurations. The logical relation is shown in [Figure 17](#). It is again a form of biorthogonal logical relation, with the main focus being the observational refinement $\mathcal{O}$: two configurations are related if they reduce to the same natural number. This coincides with what we want to show in [Theorem 3](#). To facilitate this we then lift $\mathcal{O}$ to the levels of metacontinuations, continuations and expressions. The relation $\mathcal{M}(P)$, where $P : \text{VRel}$ is a relation on values, states that two metacontinuations are related if, whenever we plug in P -related values, the resulting configurations become $\mathcal{O}$ -related. Both $\mathcal{M}$ and $\mathcal{O}$ are then used to define the relation between semantic and syntactic continuations. The relation $\mathcal{K}(Q, P)$, where $P, Q : \text{VRel}$ are relations on values, states that two continuations are related if, whenever we plug them into Q -related metacontinuations with P -related values, the resulting configurations become $\mathcal{O}$ -related. Finally, we use $\mathcal{K}$, $\mathcal{M}$, and $\mathcal{O}$ to define the relation between GITrees and λ_{delim} terms. The relation $\mathcal{E}(P, Q, R)$, where $P, Q, R : \text{VRel}$ are relations on values, states that β is related to e if, whenever we plug them into (P, Q) -related continuations and an R -related metacontinuation, the resulting configurations become $\mathcal{O}$ -related.

$$\begin{aligned}
\text{SynConf} &\triangleq \text{Expr} \times \text{Ectx} \times \text{Mcont} & \text{VRel} &\triangleq \mathbf{IT}^v \times \text{Val} \rightarrow i\text{Prop} \\
\text{SemConf} &\triangleq \mathbf{IT} \times \text{Hom} \times \text{List}(\text{Hom}) & \mathcal{O} &: \text{ConfRel} \\
\text{ConfRel} &\triangleq \text{SemConf} \times \text{SynConf} \rightarrow i\text{Prop} & \mathcal{M} &: \text{VRel} \rightarrow \text{MRel} \\
\text{MRel} &\triangleq \text{list Hom} \times \text{Mcont} \rightarrow i\text{Prop} & \mathcal{K} &: \text{VRel} \rightarrow \text{VRel} \rightarrow \text{CRel} \\
\text{CRel} &\triangleq \text{Hom} \times \text{Ectx} \rightarrow i\text{Prop} & \mathcal{E} &: \text{VRel} \rightarrow \text{VRel} \rightarrow \text{VRel} \rightarrow \text{ERel} \\
\text{ERel} &\triangleq \mathbf{IT} \times \text{Expr} \rightarrow i\text{Prop} & \llbracket \tau \rrbracket &: \text{VRel}
\end{aligned}$$

$$\begin{aligned}
\mathcal{O}((\alpha, \kappa, \sigma), (e, K, mk)) &\triangleq \text{has\_state}(\sigma) \multimap \\
&\quad \text{wp } \mathbf{P}(\kappa \ \alpha) \{ \beta. \exists v. ((e, K, mk)_{\text{eval}} \mapsto^* \langle v \rangle_{\text{ret}}) \\
&\quad \quad * (\beta, v) \in \llbracket \mathbb{N} \rrbracket * \text{has\_state}(\llbracket \beta \rrbracket) \} \\
\mathcal{M}(P)(\sigma, mk) &\triangleq \forall (\alpha, v). P(\alpha, v) \multimap \mathcal{O}((\alpha, \iota, \sigma), (v, \square, mk)) \\
\mathcal{K}(Q, P)(\kappa, K) &\triangleq \square \forall (\alpha, v). Q(\alpha, v) \multimap \forall (\sigma, mk). \mathcal{M}(P)(\sigma, mk) \multimap \\
&\quad \mathcal{O}((\alpha, \kappa, \sigma), (v, K, mk)) \\
\mathcal{E}(P, Q, R)(\beta, e) &\triangleq \forall (\kappa, K). \mathcal{K}(P, Q)(\kappa, K) \multimap \forall (\sigma, mk). \mathcal{M}(R)(\sigma, mk) \multimap \\
&\quad \mathcal{O}((\beta, \kappa, \sigma), (e, K, mk)) \\
\llbracket \mathbb{N} \rrbracket(\beta, v) &\triangleq \exists n \in \mathbb{N}. \beta = \text{Ret}(n) \wedge v = n \\
\llbracket \tau / \alpha \rightarrow \sigma / \beta \rrbracket(\theta, v) &\triangleq \exists F. \theta = \text{Fun}(F) \wedge \square \forall (\eta, w). \llbracket \tau \rrbracket(\eta, w) \multimap \\
&\quad \mathcal{E}(\llbracket \sigma \rrbracket, \llbracket \alpha \rrbracket, \llbracket \beta \rrbracket)(\theta \bullet \eta, v, w) \\
\llbracket \text{cont}(\tau, \alpha) \rrbracket(\beta, v) &\triangleq \exists \kappa K. \beta = \text{Fun}(\text{next}(\lambda x. \text{Tick}((\mathbf{P} \circ \kappa) \ x))) \wedge v = \text{cont } K \wedge \\
&\quad \mathcal{K}(\llbracket \tau \rrbracket, \llbracket \alpha \rrbracket)(\kappa, K) \\
\llbracket I \rrbracket(\rho, \gamma) &\triangleq \forall (x : \tau \in \Gamma). \square \forall \Phi. \mathcal{E}(\llbracket \tau \rrbracket, \Phi, \Phi)(\rho(x), \gamma(x)) \\
\Gamma \models_{\text{pure}} e : \tau &\triangleq \square \forall (\rho, \gamma). \llbracket I \rrbracket(\rho, \gamma) \multimap \forall \Phi. \mathcal{E}(\llbracket \tau \rrbracket, \Phi, \Phi)(\mathbf{E}[e]_{\rho}, e[\gamma]) \\
\Gamma; \alpha \models e : \tau; \beta &\triangleq \square \forall (\rho, \gamma). \llbracket I \rrbracket(\rho, \gamma) \multimap \mathcal{E}(\llbracket \tau \rrbracket, \llbracket \alpha \rrbracket, \llbracket \beta \rrbracket)(\mathbf{E}[e]_{\rho}, e[\gamma])
\end{aligned}$$

Fig. 17. Logical relation for λ_{delim} .

The relations $\mathcal{E}$ and $\mathcal{K}$ are used to give semantics $\llbracket \tau \rrbracket$ to types. The idea is that $\mathcal{E}(\llbracket \tau \rrbracket, \llbracket \alpha \rrbracket, \llbracket \beta \rrbracket)$ relates terms $\emptyset; \alpha \vdash e : \tau; \beta$ to their semantic counterparts. This is then used, as expected for logical relations, for defining the logical relation for function types and for open terms. The relation $\llbracket I \rrbracket(\rho, \gamma)$ relates the semantic environment $\rho : \text{Var} \rightarrow \mathbf{IT}$ to the syntactic substitution $\gamma : \text{Var} \rightarrow \text{Expr}$; they are related if they map the same variables to related GITrees/expressions. Then we say that an expression e is semantically valid, $\Gamma; \alpha \vdash e : \tau; \beta$, if its interpretation $\mathbf{E}[e]_{\rho}$ is related to $e[\gamma]$ under related substitutions ρ, γ . Note that if we ignore the answer types we can see that the logical relation exhibits a lot of similarities to the logical relation we gave in [Section 3.3](#), and follows the same roadmap.

For this logical relation we obtain the fundamental property, which we will use for the proof of adequacy.

Lemma 7. *Fundamental lemma.* *Let $\Gamma; \alpha \vdash e : \tau; \beta$ then $\Gamma; \alpha \models e : \tau; \beta$; and if $\Gamma \vdash_{\text{pure}} e : \tau$ then $\Gamma \models_{\text{pure}} e : \tau$.*

types	$Ty \ni \tau ::= \mathbb{N} \mid \mathbf{1} \mid \tau \rightarrow \sigma \mid \text{ref}(\tau)$	
expressions	$Expr \ni e ::= x \mid () \mid e_1 \ e_2 \mid e_1 \oplus e_2 \mid n \mid \lambda x. e$ $\mid \ell \mid \text{ref}(e) \mid !e \mid e_1 \leftarrow e_2 \mid \text{embed } e$	$\frac{\emptyset \vdash_{\text{pure}} e : \mathbb{N}}{\Gamma \vdash \text{embed } e : \mathbb{N}}$

Fig. 18. Syntax and the new typing rule of the λ_{embed} .

Proof (of Theorem 3). Note that the empty (meta)continuation is related to its denotation: $\mathcal{K}(P, P)(\text{id}, \square)$ and $\mathcal{M}(P)(\square, \square)$ hold for any relation P .

With this, we instantiate $\emptyset; \mathbb{N} \models e : \mathbb{N}; \mathbb{N}$ (that we get from Lemma 7) with the empty continuation/metacontinuation, and get the observational refinement between e and $\mathbf{E}[e]$. $\square$

This completes our treatment of denotational semantics of λ_{delim} . The next section examines interoperability of delimited continuations and other effects.

5 Modeling Interoperability Between Languages

A key advantage of using (G)Itrees for semantics is that they can provide a common framework for multi-language interaction. This section presents a case study on the interaction between the languages λ_{embed} (with higher-order store effects) and λ_{delim} (with delimited continuations). Specifically, we allow embedding closed λ_{delim} programs into λ_{embed} , and equip λ_{embed} with a type system that guarantees safe interoperability.

The embedding we provide is restrictive, preventing programs with delimited continuations from accessing outer-language continuations. We leave developing a more permissive type system for future work. At the end of the section we give an example of how to verify a more involved interaction of effects, albeit without the type system.

In this section we reuse the semantics of λ_{delim} from the previous section and higher-order store effects from Section 2. For λ_{delim} , we reuse the semantics from the previous section and higher-order store reifiers from [13]. In this section, we use **magenta** to explicitly highlight programs written in λ_{delim} , and for the interpretation functions of the denotational model of λ_{delim} .

Language λ_{embed} . λ_{embed} is a λ -calculus with base types $\mathbb{N}$ and $\mathbf{1}$, references types $\text{ref}(\tau)$ and function types $\tau \rightarrow \sigma$, with syntax given in Figure 18. Additionally, it includes a construct **embed** e for embedding λ_{delim} programs. The typing rules are all standard, except for the new typing rule for the embedding.

The idea behind the rule is that we can embed an expression from λ_{delim} if it is a “pure” expression that can evaluate to a natural number. The use of pure typing judgment for the embedded program ensures that it does not alter the answer type. This means that we can treat an embedded expression as a “complete” program, that does not require outer continuation delimiters, even though it may rely on delimited continuations internally. Those restrictions are

$\mathbf{E}[\![-]\!] : Expr \rightarrow (Var \rightarrow \mathbf{IT}) \rightarrow \mathbf{IT}$	$\mathbf{E}[\text{ref}(e)]_\rho = \text{get\_val}(\mathbf{E}[e]_\rho, \lambda x. \text{Alloc}(x, \text{Ret}))$
$\mathbf{E}[\text{embed } e]_\rho = \text{Reset}(\text{next}(\mathbf{E}[e]_\emptyset))$	$\mathbf{E}[\text{! } e]_\rho = \text{get\_val}(\mathbf{E}[e]_\rho, \lambda x. \text{Read}(x))$
$\mathbf{E}[x]_\rho = \rho(x)$	$\mathbf{E}[e_1 \leftarrow e_2]_\rho = \text{get\_val}(\mathbf{E}[e_2]_\rho, \lambda x.$
$\mathbf{E}[\ell]_\rho = \text{Ret}(\ell)$	$\text{get\_ret}(\mathbf{E}[e_1]_\rho, \lambda y. \text{Write}(y, x)))$
$\text{VRel} \triangleq \mathbf{IT}^v \rightarrow iProp$	$\llbracket \tau \rrbracket : \text{VRel}$
$\text{ERel} \triangleq \mathbf{IT} \rightarrow iProp$	$\llbracket \mathbf{1} \rrbracket(\beta) \triangleq \beta = ()$
$\mathcal{O} : \text{VRel} \rightarrow \text{ERel}$	$\llbracket \mathbf{N} \rrbracket(\beta) \triangleq \exists n. \beta = \text{Ret}(n)$
$\mathcal{O}(P)(\beta) \triangleq \text{clwp } \beta \{x. P \ x * \text{has\_state}(\llbracket \cdot \rrbracket)\}$	$\llbracket \tau \rightarrow \sigma \rrbracket(\beta) \triangleq \exists F. \beta = \text{Fun}(F) \wedge$
$\mathcal{E} : \text{VRel} \rightarrow \text{ERel}$	$\square \forall \beta. \llbracket \tau \rrbracket(\beta) \multimap$
$\mathcal{E}(P)(\beta) \triangleq \text{heap\_ctx} \multimap \text{has\_state}(\llbracket \cdot \rrbracket) \multimap$	$\mathcal{E}(\llbracket \sigma \rrbracket)(\text{Fun}(F) \bullet \beta)$
$\mathcal{O}(P)(\beta)$	$\llbracket \text{ref}(\tau) \rrbracket(\beta) \triangleq \exists \ell. \beta = \text{Ret}(\ell) \wedge$
	$\boxed{\exists \nu. \ell \mapsto \nu * \llbracket \tau \rrbracket(\nu)}$
$\llbracket \Gamma \rrbracket(\rho) \triangleq \forall (x : \tau \in \Gamma). \square \mathcal{E}(\llbracket \tau \rrbracket)(\gamma \ x)$	$\Gamma \models e : \tau \triangleq \forall \rho. \llbracket \Gamma \rrbracket(\rho) \multimap \mathcal{E}(\llbracket \tau \rrbracket)(\mathbf{E}[e]_\rho)$

Fig. 19. Denotational semantics (selected clauses) and logical relation for λ_{embed} .

crucial for the type safety of the embedding. The typing guarantees that e does not expect any additional delimiters, but it does not, by itself guarantee that any continuations in e escape the embedding boundary. To prevent that we enforce the continuation delimiter along the embedding boundary in the interpretation of embedded expressions.

Denotational model of λ_{embed} . For denotational semantics of λ_{embed} , we start by defining reifiers for the effect signature, which includes higher-order store operations (allocating, reading, and storing references) as E_{state} , and effects related to delimited continuations ($E_{\lambda_{\text{delim}}}$). Then the combined effect signature is $E_{\lambda_{\text{delim}}} \times E_{\text{state}}$, and thus we also let $\text{State} \triangleq \text{State}_{\text{delim}} \times \text{State}_{\text{state}}$, and reifiers are defined component-wise. Figure 19 shows the key parts of the denotational semantics. For most of the syntactic constructs we give the standard interpretation. For **embed** e we use the interpretation $\mathbf{E}[\![-]\!]$ for λ_{delim} from Section 4.2, and explicitly wrap the resulting GITree in a **Reset**. This continuation delimiter acts as a sort of *glue code* to protect the rest of the program from being captured by control operators from the embedded λ_{delim} program.

To show type safety of λ_{embed} , we construct a logical relation (shown in Figure 19), which is similar to the other logical relations we considered in this paper, mainly different in the observation relation $\mathcal{O}$. Given that the type system for λ_{embed} effectively prevents expressions of λ_{delim} to access contexts from λ_{embed} , we refine the observation relation to get access to a version of the **WP-HOM** rule for expression interpretations of well-typed programs of λ_{embed} , which we do not have in general, as discussed in Section 3.

Instead of the standard weakest precondition wp , we utilize a *context-local weakest precondition* clwp which bakes-in the bind rules [28].

Definition 2. *Context-local weakest precondition (clwp) is defined as follows:*
 $\text{clwp } \alpha \{ \Phi \} \triangleq \forall \kappa (\Psi : \mathbf{IT}^v \rightarrow iProp). (\forall v. \Phi \ v * \text{wp } (\kappa \ v) \{ \Psi \}) \rightarrow \text{wp } (\kappa \ \alpha) \{ \Psi \}$

Note that clwp always implies wp and validates a form of the bind rule:

$$\text{clwp } \alpha \{ \beta. \text{clwp } (\kappa \ \beta) \{ \Phi \} \} \vdash \text{clwp } (\kappa \ \alpha) \{ \Phi \}$$

for any homomorphism κ . We use the clwp in the definition of observational refinement in the model. Observational refinement asserts that if the evaluation of a top-level expression of λ_{embed} starts with an empty continuation stack, then the evaluation does not introduce new elements into the continuation stack. By using clwp we get a semantical bind lemma, which can be seen as a version of [WP-HOM](#) for semantically valid expressions of λ_{embed} . As before, we then obtain fundamental lemma and denotational type soundness.

Lemma 8. *Semantical bind.* $\mathcal{E}(\lambda x : \mathbf{IT}^v. \mathcal{E}(P)(\kappa \ x))(\beta)$ implies $\mathcal{E}(P)(\kappa \ \beta)$ for any homomorphism κ .

Lemma 9. *Fundamental lemma.* Let $\Gamma \vdash e : \tau$ then $\Gamma \models e : \tau$.

Lemma 10. *Denotational type soundness.* Let $\emptyset \vdash e : \tau$ and $(\mathbf{E}[e]_{\emptyset}, []) \rightsquigarrow^* (\alpha, \sigma)$, then $(\exists \beta \ \sigma'. (\alpha, \sigma) \rightsquigarrow (\beta, \sigma')) \vee (\alpha \in \mathbf{IT}^v)$.

Unrestricted interaction of delimited continuations and higher-order state. Even though the type system we considered here is restrictive, we can still reason about unrestricted interactions of events in the “untyped” setting. Here we show an example of such an unrestricted interaction, and demonstrate how to reason about context-dependent and context-independent effects at the same time. While this kind of interactions is forbidden by our type system, we can still write and prove meaningful specifications for such programs. Consider the program in [Figure 20](#), written in GITrees directly. The function `prog` utilizes both delimited continuations and state. It takes a reference y as its argument and begins by allocating the value 1 in the store at reference x . Then it captures the continuation `get_ret(y, ( $\lambda l$ . Let  $p = \text{NatOp}_+(\text{Read}(l), \dots)$  in  $\text{Write}(l, p)$ ))` as k . Invoking the continuation k with a number n increments the current value of y by n . The program then invokes this continuation twice. First with the original value of x . Then, with an incremented value of x . Since the starting value of x is 1, the reference y is incremented first by 1 and then by 2. We capture this behavior in the specification for `prog` stated in [Figure 20](#).

It is important to note that this program features a bidirectional interaction between state and continuations. Specifically, the body of `Shift` involves state operations, while the result of `Shift` is subsequently used to increment a value in the heap. As we have seen, while this type of interaction is not allowed in the type system, we can still reason about them in program logic. We stipulate that our proposed type system could potentially be extended to support embedding “pure” functions, allowing for bi-directional interaction between the two languages. We believe that such an extension would require implementing answer-type polymorphism, following the approach of Asai and Kameyama [2].

<pre> prog $\triangleq$ Fun(next(λy. Let $x = \text{Alloc}(\text{Ret}(1))$ in Let $n = \text{Shift}(\lambda k. \text{next}(\text{Appcont}'(\text{Read}(x), k)$; Let $m =$ NatOp$_{+}(\text{Read}(x), \text{Ret}(1))$ in Write(x, m) ; Appcont'$(\text{Read}(x), k))$ in get_ret($y, (\lambda l. \text{Let } p =$ NatOp$_{+}(\text{Read}(l), n)$ in Write(l, p)))))) </pre>	Initial offset value Capture continuation as k First call to k with the init. value of x $x := x + 1$; Second call to k with updated x

where Appcont' $(x, y) \triangleq \text{Appcont}(\text{next}(x), \text{next}(\text{Tau} \circ y \circ \text{next}))$

$$\frac{\text{heap\_ctx} \quad \text{has\_state}(\sigma) \quad y \mapsto \text{Ret}(n)}{\text{wp}(\text{Reset}(\text{next}(\text{prog} \bullet \text{Ret}(y)))) \{y \mapsto \text{Ret}(n+3) * \text{has\_state}(\sigma)\}}$$

Fig. 20. Example program with delimited continuations and state and its specification.

6 Discussion and Related Work

We conclude the paper by discussing related work and future directions.

This paper extends GITrees to handle context-dependent effects, which allows us to model higher-order languages with control operators like `call/cc` ($x. e$) and `S` $x. e$. We showed this extension supports interoperability between languages with different context-dependent effects, while preserving reasoning about context-independent effects. Our approach leverages the native support for higher-order functions and effects in GITrees. This differs from the first-order effect representation of effects in ITrees [34], which would require explicitly first-order representation of functions and continuations, if we want to model first-class continuation. Such model would mix syntactic and semantic concerns, which is part of what we are trying to avoid by working with (G)ITrees.

Another difference with ITrees is our approach to reasoning. While ITrees use bisimulation-based equational theory, we follow GITrees in using tailored program logics and defining refinements. Our logic are expressive enough to define logical relations and carry out computational adequacy proofs. In future work, it would be interesting to develop techniques for reasoning about weaker notions of equality than the basic equational theory that GITrees comes equipped with, see the more extensive discussion of this point in [13].

The “classical” domain theory remains an important source of inspiration and ideas for our development, and we want to mention some of the related work along those lines. Cartwright and Felleisen [9] introduced a framework of extensible direct models for constructing modular denotational semantics of programming languages. Their framework centers on an abstract notion of *resources* for representing effects (such as store or continuations) and a central `admin` function that manages these resources. Each language extension defines both the types (domains) of additional values and resources, and specifies the *actions* that the `admin` function can perform on these resources. Building on this framework,

Sitaram and Felleisen [26] demonstrated that such models can provide direct-style fully abstract semantics for control operators. Their approach interprets effects, including continuations, by delegating them to a top-level handler. Our work adopts several key ideas from this line of research but reformulates them in the context of GITrees rather than classical domain theory. In our framework, effect signatures define resources, reifiers specify actions, and the reduction relation serves as the central authority dispatching the effects. The transition to GITrees enables us to formalize the extensibility of this approach in a practical manner, and it allows us to develop program logics where “resources” (as above) become resources in separation logic.

Compared to other programming languages paradigms and effects, type systems and logical relations for delimited continuations have not been studied as comprehensively. The original type system for **shift/reset** is due to Danvy and Filinski [10], where they employ *answer-type modification*. Materzok and Biernacki [22] generalized this type system to account for more involved control operators **shift₀/reset₀**; an alternative substructural type system for these operators was designed by Kiselyov and Shan [17]. Dyvbig, Peyton Jones, and Sabry [11] provide a typed monadic account of CPS for delimited continuations with dynamic prompt generation. Asai and Kameyama [2] present a polymorphic variant of the Danvy and Filinski’s type system.

Biernacka and Biernacki [5] prove termination for a language with $\mathcal{S} \ x. e$ and $\mathcal{D} \ e$ (but without recursion) using logical relations based on abstract machine semantics. The shape of their logical relation is similar to our logical relation used for showing adequacy in that they also have relations for configurations, metacontinuations, etc. Asai [1] uses type-directed logical relations to verify a direct-style specializer (partial evaluator) for a language with $\mathcal{S} \ x. e$ and $\mathcal{D} \ e$, proving correctness against evaluation-context based operational semantics. In contrast with those works, we define our logical relations on denotations, using the semantics of GITrees and the derived program logics.

In our interoperability example we showed type safety of a combined language, with respect to denotational semantics. In future work we would like to examine other properties: for example, that λ_{delim} expressions cannot disrupt λ_{embed} ’s control flow, perhaps establishing some form of well-bracketedness as in [29].

We would also like to study other context-dependent effects like exceptions, handlers, and algebraic effects [24,4,33,31,3]. In particular, it would be interesting to give a denotational semantics to a language with handlers, derive program logic rules for it, and compare the resulting program logic to the one in [32].

Acknowledgments. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

Data-Availability Statement. The formalisation associated with this paper, which covers all the main results, is available online [27].

References

1. Asai, K.: Logical relations for call-by-value delimited continuations. In: van Eekelen, M.C.J.D. (ed.) *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*. Trends in Functional Programming, vol. 6, pp. 63–78. Intellect (2005)
2. Asai, K., Kameyama, Y.: Polymorphic Delimited Continuations. In: Shao, Z. (ed.) *Programming Languages and Systems*. pp. 239–254. Springer. https://doi.org/10.1007/978-3-540-76637-7_16
3. Bach Poulsen, C., van der Rest, C.: Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proceedings of the ACM on Programming Languages* **7**(POPL), 62:1801–62:1831 (Jan 2023). <https://doi.org/10.1145/3571255>
4. Bauer, A., Pretner, M.: Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* **84**(1), 108–123 (Jan 2015). <https://doi.org/10.1016/j.jlamp.2014.02.001>
5. Biernacka, M., Biernacki, D.: Context-based proofs of termination for typed delimited-control operators. In: Porto, A., López-Fraguas, F.J. (eds.) *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. pp. 289–300. ACM (2009). <https://doi.org/10.1145/1599410.1599446>, <https://doi.org/10.1145/1599410.1599446>
6. Biernacka, M., Biernacki, D., Danvy, O.: An operational foundation for delimited continuations in the CPS hierarchy. *Log. Methods Comput. Sci.* **1**(2) (2005). [https://doi.org/10.2168/LMCS-1\(2:5\)2005](https://doi.org/10.2168/LMCS-1(2:5)2005), [https://doi.org/10.2168/LMCS-1\(2:5\)2005](https://doi.org/10.2168/LMCS-1(2:5)2005)
7. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. *Proc. ACM Program. Lang.* **3**(POPL) (jan 2019). <https://doi.org/10.1145/3290319>, <https://doi.org/10.1145/3290319>
8. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* **8**(4) (2012). [https://doi.org/10.2168/LMCS-8\(4:1\)2012](https://doi.org/10.2168/LMCS-8(4:1)2012), [https://doi.org/10.2168/LMCS-8\(4:1\)2012](https://doi.org/10.2168/LMCS-8(4:1)2012)
9. Cartwright, R., Felleisen, M.: Extensible denotational language specifications. In: Hagiya, M., Mitchell, J.C. (eds.) *Theoretical Aspects of Computer Software*. pp. 244–272. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-57887-0_99
10. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. Tech. rep., DIKU – Computer Science Department, University of Copenhagen (1989)
11. Dyvbig, R.K., Peyton Jones, S., Sabry, A.: A monadic framework for delimited continuations **17**(6), 687–730. <https://doi.org/10.1017/S0956796807006259>, <https://www.cambridge.org/core/journals/journal-of-functional-programming/article/monadic-framework-for-delimited-continuations/D99D1394370DFA8EA8428D552B5D8E7E>
12. Felleisen, M., Friedman, D.P.: Control operators, the secd-machine, and the λ -calculus. In: Wirsing, M. (ed.) *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. pp. 193–222. North-Holland (1987)
13. Frumin, D., Timany, A., Birkedal, L.: Modular denotational semantics for effects with guarded interaction trees. *Proc. ACM Program. Lang.* **8**(POPL), 332–361 (2024). <https://doi.org/10.1145/3632854>, <https://doi.org/10.1145/3632854>

14. Iris team: The Iris Project website and Coq development (2025), <https://iris-project.org/>
15. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* **28**, e20 (2018). <https://doi.org/10.1017/S0956796818000151>, <https://doi.org/10.1017/S0956796818000151>
16. King, A.: Delimited continuation primops. <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst> (2021), accessed: 2024-06-27
17. Kiselyov, O., Shan, C.c.: A Substructural Type System for Delimited Continuations. In: Della Rocca, S.R. (ed.) *Typed Lambda Calculi and Applications*. pp. 223–239. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73228-0_17
18. Koh, N., Li, Y., Li, Y., Xia, L.y., Beringer, L., Honoré, W., Mansky, W., Pierce, B.C., Zdancewic, S.: From C to interaction trees: Specifying, verifying, and testing a networked server. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 234–248. CPP 2019, Association for Computing Machinery, New York, NY, USA (Jan 2019). <https://doi.org/10.1145/3293880.3294106>
19. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Castagna, G., Gordon, A.D. (eds.) *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017*. pp. 205–217. ACM (2017). <https://doi.org/10.1145/3009837.3009855>, <https://doi.org/10.1145/3009837.3009855>
20. Leijen, D.: Koka: Programming with row polymorphic effect types. In: Levy, P.B., Krishnaswami, N. (eds.) *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. EPTCS*, vol. 153, pp. 100–126 (2014). <https://doi.org/10.4204/EPTCS.153.8>, <https://doi.org/10.4204/EPTCS.153.8>
21. Leijen, D.: Structured asynchrony with algebraic effects. In: Lindley, S., Yorgey, B.A. (eds.) *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*. pp. 16–29. ACM (2017). <https://doi.org/10.1145/3122975.3122977>, <https://doi.org/10.1145/3122975.3122977>
22. Materzok, M., Biernacki, D.: Subtyping delimited continuations **46**(9), 81–93. <https://doi.org/10.1145/2034574.2034786>, <https://doi.org/10.1145/2034574.2034786>
23. Pitts, A.M.: Typed operational reasoning. In: Pierce, B.C. (ed.) *Advanced Topics In Types And Programming Languages*, chap. 7, pp. 245–289. The MIT Press (2004)
24. Plotkin, G.D., Pretnar, M.: Handling Algebraic Effects. *Logical Methods in Computer Science* **Volume 9, Issue 4** (Dec 2013). [https://doi.org/10.2168/LMCS-9\(4:23\)2013](https://doi.org/10.2168/LMCS-9(4:23)2013)
25. Silver, L., He, P., Cecchetti, E., Hirsch, A.K., Zdancewic, S.: *Semantics for Noninterference with Interaction Trees* (2023)
26. Sitaram, D., Felleisen, M.: Models of continuations without continuations. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 185–196. POPL '91, Association for Computing Machinery. <https://doi.org/10.1145/99583.99611>, <https://dl.acm.org/doi/10.1145/99583.99611>
27. Stepanenko, S., Nardino, E., Frumin, D., Timany, A., Birkedal, L.: Context-dependent effects in guarded interaction trees (Jan 2025). <https://doi.org/10.5281/zenodo.14623650>, <https://doi.org/10.5281/zenodo.14623650>

28. Timany, A., Birkedal, L.: Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* **3**(ICFP), 105:1–105:28 (2019). <https://doi.org/10.1145/3341709>, <https://doi.org/10.1145/3341709>
29. Timany, A., Guéneau, A., Birkedal, L.: The logical essence of well-bracketed control flow. *Proc. ACM Program. Lang.* **8**(POPL), 575–603 (2024). <https://doi.org/10.1145/3632862>, <https://doi.org/10.1145/3632862>
30. Timany, A., Krebbers, R., Dreyer, D., Birkedal, L.: A logical approach to type soundness. *Journal of the ACM* **71** (2024)
31. van den Berg, B., Schrijvers, T., Poulsen, C.B., Wu, N.: Latent Effects for Reusable Language Components. In: Oh, H. (ed.) *Programming Languages and Systems*. pp. 182–201. *Lecture Notes in Computer Science*, Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-89051-3_11
32. de Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. *Proc. ACM Program. Lang.* **5**(POPL) (jan 2021). <https://doi.org/10.1145/3434314>, <https://doi.org/10.1145/3434314>
33. Wu, N., Schrijvers, T., Hinze, R.: Effect handlers in scope. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. pp. 1–12. Haskell ’14, Association for Computing Machinery, New York, NY, USA (Sep 2014). <https://doi.org/10.1145/2633357.2633358>
34. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.* **4**(POPL), 51:1–51:32 (2020). <https://doi.org/10.1145/3371119>, <https://doi.org/10.1145/3371119>
35. Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages* **5**(ICFP), 67:1–67:30 (Aug 2021). <https://doi.org/10.1145/3473572>
36. Zhang, H., Honoré, W., Koh, N., Li, Y., Li, Y., Xia, L.Y., Beringer, L., Mansky, W., Pierce, B., Zdancewic, S.: Verifying an HTTP Key-Value Server with Interaction Trees and VST. In: Cohen, L., Kaliszyk, C. (eds.) *12th International Conference on Interactive Theorem Proving (ITP 2021)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 193, pp. 32:1–32:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). <https://doi.org/10.4230/LIPIcs.ITP.2021.32>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





An Automata-theoretic Basis for Specification and Type Checking of Multiparty Protocols

Felix Stutz^{1,2(✉)}  and
Emanuele D'Ossualdo^{1,3} 

¹ MPI-SWS, Saarbrücken, Germany
`felix.stutz@uni.lu`

² University of Luxembourg, Esch-sur-Alzette, Luxembourg
`emanuele.dosualdo@uni-konstanz.de`

³ University of Konstanz, Konstanz, Germany

Abstract. We propose the Automata-based Multiparty Protocols framework (AMP) for top-down protocol development. The framework features a new very general formalism for global protocol specifications called Protocol State Machines (PSMs), Communicating State Machines (CSMs) as specifications for local participants, and a type system to check a π -calculus with session interleaving and delegation against the CSM specification. Moreover, we define a large class of PSMs, called “tame”, for which we provide a sound and complete PSPACE projection operation that computes a CSM describing the same protocol as a given PSM if one exists. We propose these components as a backwards-compatible new backend for frameworks in the style of Multiparty Session Types. In comparison to the latter, AMP offers a considerable improvement in expressivity, decoupling of the various components (e.g. projection and typing), and robustness (thanks to the complete projection).

Keywords: Communication protocols · Verification · Multiparty session types · Communicating state machines · Type checking

1 Introduction

Designing correct distributed communication protocols is an important and hard problem. Consider a finite set of protocol *participants* (i.e. independent processes) whose only means of interaction between each other is asynchronous message passing through reliable FIFO channels. The goal is to program each participant so that some global emergent behaviour is achieved, e.g. a leader is elected. Unfortunately, even when each participant is finite-state, the presence of unbounded delays (i.e. unbounded communication channels) makes any non-trivial property of the emergent global behaviour undecidable [10].

The top-down protocol design approach proposes to work around this issue by a reversal in the methodology: instead of first programming the participants and then checking that their global behaviour is what we desired it to be, we first specify the desired global behaviour and then synthesize each participant’s local

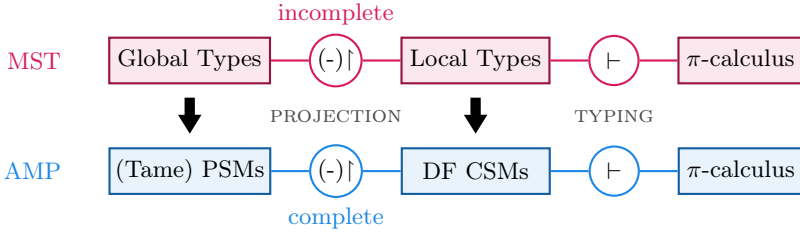


Fig. 1: The components of top-down frameworks.

specification so that local behaviour gives rise to the correct global behaviour by construction. Each participant’s concrete implementation is then checked against its local specification, which (a) can be achieved by static means like type systems, and (b) makes the verification of the implementation local and modular.

Multiparty Session Types (MSTs) [37] is one of the most prominent and extensively studied formalisms supporting this top-down design methodology. The key components of the framework, depicted in Fig. 1, are: (1) **Global Types**: a dedicated language to specify correct global behaviour; (2) **Local Types**: a dedicated language to specify each participant’s actions in the protocol; (3) **Programs**: a programming language (typically a π -calculus) to express concrete implementations of each participant of the protocol.

Imagine, as a simple example, we want to specify a centralised leader election protocol, where an arbiter a selects a leader among p and q and the selected participant communicates the win to the other. A possible global type representing the protocol is $G = (a \rightarrow p:\text{sel}. p \rightarrow q:\text{win}) + (a \rightarrow q:\text{sel}. q \rightarrow p:\text{win})$ where $a \rightarrow q:\text{sel}$ says that a sends sel and q receives it and $+$ denotes branching. The local type of p would then be $(p \triangleleft a?\text{sel}. p \triangleright q!\text{win}) + p \triangleleft q?\text{win}$ where $p \triangleleft a?\text{sel}$ means “ p receives message sel from a ” and $q \triangleleft p?\text{win}$ means “ p sends message win to q ”. Thus, p is supposed to listen for a message from a or from q ; in the former case it would then communicate the win to q , in the latter, just concede. A program implementation may consist of a process for each participant; the process for the arbiter a may implement any specific policy for selecting the leader (e.g. always choose p), as long as the communications follow the protocol.

The relationship between the three representations of the protocol, i.e. global types, local types, and programs, is delicate. First, the global type and the local types should give rise to the same behaviour; however it is not always possible to capture the behaviour of a global type with local types. Suppose, for instance, that we modified the leader election protocol G to $G' = (a \rightarrow p:\text{sel}. q \rightarrow p:\text{lose}) + (a \rightarrow q:\text{sel}. p \rightarrow q:\text{lose})$. While, from a global perspective, it is possible to insist on the losing participant informing the winner that they lost, locally, the losing participant has no way to determine whether they won or not. Therefore G' is not realisable by local processes: we say it is not projectable. Second, local types are a more abstract representation of the system than programs, but we still want to show that, when implementation details are omitted, a program only performs communications that adhere to the local specification.

In MSTs, the relationship between the three layers of the framework are enforced through two procedures: (i) **Projection**, which (when possible) extracts, from a global type G , some local types that are guaranteed to behave as described by G ; and (ii) **Type Checking**, which checks that the program implementation of each participant adheres to the behaviour specified in its local type.

In a perfect world, a framework for top-down protocol design should be:

1. **Expressive:** It should support as many protocols as possible.
2. **Decoupled:** Its components (global/local specifications, programs, projection, type checking) should depend on each other as little as possible, and be specified independently of their algorithmic implementation, to allow for reuse and modularity.
3. **Robust:** It should only reject a global specification if there is a genuine issue with it (i.e. no false positives).

Unfortunately, the MST frameworks in the literature leave something to be desired against this ideal picture. They all suffer from:

- **Expressivity Limitations:** Many legitimate protocols are rejected either because the global specification syntax is too restricted, or because the projection algorithm cannot handle them. For example, every MST framework we are aware of can only handle global types with *directed choice*, i.e. where every branching point involves exactly one sender and one receiver. This immediately rules out our example leader election protocol G because the branches involve different receivers.
- **Decoupling Limitations:** In MSTs, the syntax of global types directly influences the definition of the projection algorithm and the syntax of the local types, which in turn influence the type system design. Typically, changing one of the framework’s components requires adapting (and reproving correctness of) all the others. Furthermore, many MST frameworks solely give the intended relation between global and local types through the projection algorithm and do not give a declarative definition.
- **Robustness Limitations:** The heuristic nature of the projection algorithms makes it very hard to predict if a global type will be handled or not by an MST framework, even in the case where the behaviour specified by the global type is unproblematic.

In this paper, we propose a new foundation for top-down protocol design machinery, dubbed AMP (Automata-based Multiparty Protocols), that achieves the expressivity, decoupling and robustness goals.

Expressivity of AMP. Figure 1 shows the components of AMP. To achieve expressivity, we propose a new general formalism for (finite-control) global protocol specifications, which we call *Protocol State Machines* (PSMs). The formalism is based on automata which are given semantics in terms of (finite and infinite) sets of words, over an alphabet of send ($p \triangleright q!m$) and receive ($q \triangleleft p?m$) actions. PSMs remove many of the restrictions of global types, while retaining their character: they specify the expected behaviour from a global perspective, and satisfy some basic correctness properties by construction (e.g. every send is eventually received, no type mismatches, etc). Owing to their generality, PSMs can represent

any global type, but can go well beyond them: they also strictly subsume High-Level Message Sequence Charts (HMSCs). For maximizing expressivity at the local level, we adopt Communicating State Machines (CSMs) as the formalism for local protocol specifications. They are a canonical representation for decentralised asynchronous communication and as such do not impose constraints over what can be represented. Finally, to maximise expressivity at the program level, we consider a π -calculus with session interleaving and delegation.

Decoupling of AMP. In AMP, decoupling is achieved through its handling of projection and type checking. For projection, the framework merely specifies the semantic requirements that a correct projection algorithm needs to satisfy: essentially that it produces a deadlock-free CSM which represents the same language as the input PSM. This limits the impact of projection on global and local specifications, and leaves open any algorithmic/manual strategy to achieve the projection goal. (We discuss how AMP proposes to actually implement projection in the discussion of robustness). For example, scenarios in which the user provides both the PSM and the CSM and a proof that they represent the same language and the CSM satisfies desirable properties (like deadlock freedom) or where an algorithm infers a PSM from a CSM, are both compatible with our framework thanks to this declarative approach to projection. This treatment is in line with some MST works where the fundamental property of projection is expressed in terms of some behavioural equivalence between local and global types. For type checking, decoupling is achieved by defining the type system by depending exclusively on programs and CSMs. The standard guarantees of subject reduction, communication safety and session fidelity are proven by appealing to properties of CSMs. This demonstrates how effective CSMs are in providing a clean decoupled interface between projection and type checking.

Robustness of AMP. Finally, we demonstrate how robustness can be achieved in AMP, by identifying a large class of PSMs, called *Tame PSMs*, for which we provide a decidable, sound and complete projection operation. Tame PSMs extend the reach of sound and complete projection beyond global types and can handle a large class of HMSCs as well as protocols that cannot be expressed as either global types nor HMSCs. The main constraint that makes a PSM tame is what we call *sender-driven choice*: that at any branching point, the sender in all the branches is the same participant and takes distinct actions in the branches. Our projection algorithm builds on a recently proposed complete projection for sender-driven global types [48]. Thanks to a surprising reduction, we manage to extend the algorithm to tame PSMs while keeping the complexity in PSPACE. Due to the fact that our projection operation is complete, only protocols that do not admit any valid projection will be rejected: those are protocols which simply cannot be implemented by local processes. We also show that our class is in a sense “maximally robust”: lifting the sender-driven restriction makes projection undecidable, even for global types. AMP is also robust in the sense that one can select the desired guarantees of the type system and check whether they can be enforced by checking (syntactic) properties of the global protocol, pinpointing exactly which guarantee is provided by a PSM. Finally, we show that the frame-

work is backwards-compatible with MSTs: not only can we encode global types into PSMs and project them, we also pinpoint the (simple) conditions under which our projection yields CSMs which are equivalent to local types.

Contributions. In summary:

- We propose PSMs as an expressive general formalism for (finite-control) global protocol specifications.
- We propose CSMs as a canonical model for local protocol specifications and specify their desired relationship with PSMs declaratively.
- We define the first session type system based on CSMs, pinpointing exactly the properties of the CSM that are needed to provide each of the desired guarantees; these properties can be enforced by construction by ensuring the PSMs conform to some simple checks.
- We define Tame PSMs (encompassing all directed and sender-driven global types) and give a sound and complete projection algorithm for them.
- We show that sender-driven choice is a necessary restriction even for global types: projection is undecidable otherwise.
- We characterise which class of PSMs corresponds to global types, and which CSMs correspond to local types, giving us full backward-compatibility with standard MST theory.

We think of AMP as a backend for top-down protocol design tools with the following workflow. Any specific tool, e.g. Scribble [71], provides a dedicated syntax for types and processes. Then, a global specification is compiled to a PSM (where the compiler guarantees its tameness, which would be trivial for global types) and invokes the projection of AMP, producing a CSM. This could be re-translated for user consumption, but also be used to drive typing using AMP. Failure of projection can be directly translated by the frontend to an explanation of why the protocol is flawed and must be repaired. Given the generality of PSMs, it should also be easier to experiment with extensions of the frontend language.

All proofs, omitted details, and additional examples can be found in [62].

2 Motivation and Key Ideas

In this section, we give an informal overview of the key ideas behind AMP before proceeding with the formal development from Section 3.

2.1 Global Specifications via Protocol State Machines

Our first goal is to define an expressive formalism for specifying global protocols, that is also constrained enough to make it tractable for top-down protocol development. One of the most accomplished such formalisms, used in MSTs, is *global types*. Figure 2 shows an example of a global type, represented in Figure 3 as an HMSC.

The term $p \rightarrow q : m_1$ indicates the transmission of message m_1 from p to q . The symbol $\mathbf{0}$ denotes termination of the protocol. Recursion can be specified

by binding a recursion variable X with μX and using X subsequently to jump back to where X was bound. Branching is denoted by $+$. In the example, p can pick between three branches by sending different messages to q . Subsequently, q sends messages to r in all branches: 1 in the top and middle branch and 3 in the bottom branch. Participant r is supposed to send messages v_1 or v_2 in the top and middle branch while it receives from p in the bottom branch, which also recurses using X .

What makes these formalisms tractable? Their first key characteristic is that, as a specification tool, they allow the user to (a) adopt a global point of view, describing what coordination between all the participants is induced by the protocol; (b) express this coordination without enumerating all possible interleavings of the send and receive events that can happen due to the asynchronous nature of communication, e.g. $p \rightarrow q; m_1$ indicates the send of the message immediately followed by its receipt, although in any asynchronous implementation, the receipt might happen at a much later point, after other independent events took place. In Fig. 2, r may lag behind arbitrarily while p and q keep sending messages. The second key characteristic of global types and HMSCs is that they are *finite-control*: their control structure can be described using a finite graph. This makes it possible to algorithmically manipulate them, e.g. for verifying they satisfy some desirable properties, or for extracting local protocol specifications.

Our aim is to distil these two characterising features and remove any other restriction that is not necessary, to obtain a more expressive global specification formalism. To do this, we take a language-theoretic view of protocols, where a protocol is seen as the set of sequences of send and receive events that are considered compliant with it. More precisely, a send event $p \triangleright q!m$ records that p sent the message m to q ; a receive event $q \triangleleft p?m$ records that q received message m from p . A protocol specification is the language of desired finite or infinite words of events. For the purpose of this section, we will focus on finite words, but the technical development considers both finite and infinite words.

Not all languages over these events are meaningful in the context of protocols. First, the sequences of events might not be feasible when using FIFO channels (e.g. $p \triangleright q!1 \cdot q \triangleleft p?2$ is not FIFO); we write FIFO for the language of all words that satisfy FIFO order. Second, if $p \triangleright q!m_1 \cdot r \triangleright q!m_2 \cdot q \triangleleft p?m_1 \cdot q \triangleleft r?m_2$ is accepted by a protocol, it ought to also accept $r \triangleright q!m_2 \cdot p \triangleright q!m_1 \cdot q \triangleleft p?m_1 \cdot q \triangleleft r?m_2$ as this kind of reorderings are induced by the scheduling of participants and network delays which are out of the control of participants. We write $\mathcal{C}(L)$ for the closure of the language L under such reorderings. Thus a language $L \subseteq \text{FIFO}$ represents the global interaction patterns of the protocols; moreover L can specify only some

$$\mu X. + \left\{ \begin{array}{l} p \rightarrow q; m_1 . q \rightarrow r; 1 . r \rightarrow p; v_1 . 0 \\ p \rightarrow q; m_2 . q \rightarrow r; 1 . r \rightarrow p; v_2 . 0 \\ p \rightarrow q; m_3 . q \rightarrow r; 3 . p \rightarrow r; v_3 . X \end{array} \right.$$

Fig. 2: Example global type.

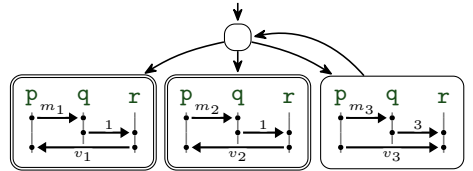


Fig. 3: A protocol as an HMSC.

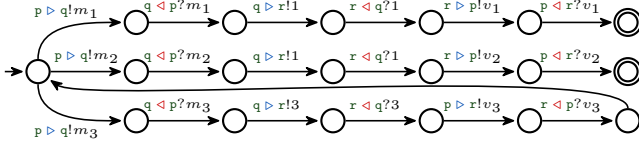


Fig. 4: A PSM encoding for the protocol of Fig. 2.

of these interactions and get all the ones that should also be possible under the asynchronous semantics by declaring the full set of acceptable words to be $\mathcal{C}(L)$.

Now, to obtain a finite-control formalism, we propose to express such a “core” language L for the protocol $\mathcal{C}(L)$ using a finite state machine M with $\mathcal{L}(M) = L$. Since FIFO is not regular, the only feasible way of ensuring $\mathcal{L}(M) \subseteq \text{FIFO}$ is by requiring M to keep track of which sent messages are still pending, which a finite-state machine can only do up to some maximum capacity for the send buffers. We thus arrive at the requirement that $\mathcal{L}(M) \subseteq \text{FIFO}_B$, where FIFO_B is the set of words respecting FIFO but where the number of pending sends never exceed $B \in \mathbb{N}$ at any point in time. Note that FIFO_B is regular.

Building on these observations, we define a *Protocol State Machine* (PSM) to be a finite state machine M recognising words of send and receive events, with $\mathcal{L}(M) \subseteq \text{FIFO}_B$ for some $B \in \mathbb{N}$. Fig. 4 shows the protocol of Fig. 2 as a PSM. Interpreted as a mere automaton M , it recognises a language $\mathcal{L}(M)$ of words with at most one pending send at all time. (We call M $\Sigma 1$ -PSM because the total number of messages in flight is at most 1; if we allowed 1 message *per channel*, it would be called a 1-PSM.) As a PSM, however, M denotes the language $\mathcal{C}(\mathcal{L}(M))$, which admits words with unbounded channel behaviours and is not even regular in general. For instance, $\mathcal{C}(\mathcal{L}(M))$ includes words starting with $(p \triangleright q!m_3 \cdot q \triangleleft p?m_3 \cdot q \triangleright r!3 \cdot p \triangleright r!v_3)^n \cdot (r \triangleleft q?3 \cdot r \triangleleft p?v_3)^n \dots$ where r is running at a lower rate than the other participants, and leaves n pending sends from p and from q before it consumes them.

PSMs achieve a substantial gain in expressivity while retaining the key characteristics of global types. In terms of expressivity, every global type can be encoded as a $\Sigma 1$ -PSM; furthermore PSMs can be used to encode HMSCs, which strictly subsume global types because the latter cannot specify simultaneous message exchanges between a pair of participants [63]. PSMs can even represent protocols that are outside the reach of HMSCs. Consider, for example, the PSM in Fig. 5. In that protocol, p commits to some integer (abstracted as the label int) at the beginning by sending it to r and sends a go signal to q . Note that here we use the paired send and receive notation $p \rightarrow q:\text{ok}$ to emit the two events in sequence. Then q and r engage in some negotiation of arbitrary length until q decides to exit the loop, at which point r is finally allowed to receive the message sent by p . No HMSC can represent such protocol: the matching events $p \triangleright r!\text{int}$ and $r \triangleleft p?\text{int}$ are separated by an arbitrary number of events (with no opportunity for reordering up to $\mathcal{C}(-)$); since matching events in HMSCs need to belong to the same basic block, such block would also need to contain the arbitrarily many events in between, which is impossible.

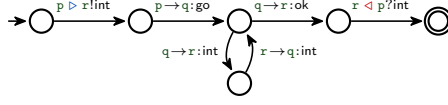


Fig. 5: A protocol not expressible as an HMSC. Transitions labelled with $p \rightarrow q:m$ should be interpreted as emitting the sequence $p \triangleright q!m \cdot q \triangleleft p?m$.

Of course, this level of generality would be pointless if we were not able to provide for it in the other components of top-down protocol design. We start by studying the first crucial component: projection.

2.2 From Global to Local Specifications: Projection

When considering projection, our first concern is the goal of decoupling: we want to define a general interface for projection, such that both different algorithmic implementations of projection can be used without altering the design of the rest of the framework; and such that typing is not dependent on global specifications (nor projection details).

In AMP, the key to decoupling is in choosing *Communicating State Machines* (CSMs) as the formalism for local specifications. A CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ associates a finite state automaton A_p to each participant $p \in \mathcal{P}$, where transitions can either send or receive on the channels of p ; the semantics of $\{\{A_p\}_{p \in \mathcal{P}}\}$ is defined on configurations that include the local states for each participant and an (unbounded) FIFO buffer for each channel. They induce a FIFO language $\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\})$ over send/receive events, by considering as final the configurations where all the participants are in final local states and all the buffers are empty. CSMs thus represent a canonical general model of finite-control asynchronous protocol implementations.

Per se, this is not a particularly original choice: MST's local types have been linked to CSMs of a certain shape before [23, 60], and HMSC-based work used them as local specifications. What AMP demonstrates is that it is possible to build the entire top-down methodology around CSMs (with fewer restrictions), including a session type system, gaining both in expressivity and in decoupling.

Having fixed our model for local behaviour, we can ask when it defines behaviour consistent with a global specification. We say a CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ is a *projection* of a PSM M if $\{\{A_p\}_{p \in \mathcal{P}}\}$ is deadlock-free and $\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}) = \mathcal{C}(\mathcal{L}(M))$.

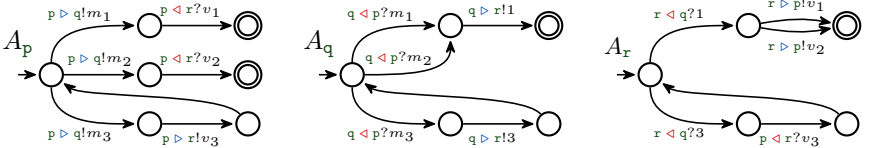


Fig. 6: Example CSM.

We focus on the (*projection*) *synthesis problem*, producing a CSM as result. The corresponding decision problem is the *projectability problem*, which simply asks if there exists such a CSM. Notably, projectability can have lower complexity.

Even for simple protocols, projection can be tricky. Take the example of Fig. 2: r can never distinguish between the top two branches, as its only observations would be to have received 1 from q . The instance of the protocol with $m_1 \neq m_2$ and $v_1 \neq v_2$ would thus not be projectable. If $m_2 = m_3$ then q would not be able to send the appropriate message to r . Therefore, the only projectable instances with no redundant branches are the ones where m_1 , m_2 , and m_3 are pair-wise distinct and $v_1 = v_2$. Figure 6 shows a candidate projection of the PSM in Fig. 2. If $m_2 = m_3$ or $v_1 \neq v_2$, the PSM is not projectable, and in fact the CSM can reach a deadlock.

Given CSMs are Turing-complete models, it is unsurprising that checking if a given CSM is a projection of a given PSM is undecidable. The key advantage of the top-down approach boils down to the fact that it is nevertheless often possible to efficiently *compute* a valid projection from a global specification. This is precisely the goal of the *projection* operation. A projection operation $(-)\downarrow$ is a function taking a PSMs as input and returning either $\perp$ or a CSM; it is a *sound projection* if for all PSM M , if $M\downarrow = \{\{A_p\}_{p \in \mathcal{P}}\}$ then $\{\{A_p\}_{p \in \mathcal{P}}\}$ is a projection of M ; it is a *complete projection* if for all projectable M , $M\downarrow \neq \perp$.

The MST literature proposed a number of sound but incomplete projection algorithms for global types. Incompleteness makes MST frameworks lack robustness: a projectable global type might still be rejected by the framework because the projection is unable to handle it; this leaves the user in the awkward position of having to build a mental model for the projection algorithm to be able to design viable global types. Li et al. [48] proposed the first sound and complete projection algorithm for sender-driven global types. Its PSPACE complexity stems from the need for determinisation. Their evaluation, though, showed that these corner cases will likely not occur in reality. This provides initial evidence that robustness is achievable without compromising efficiency.

As is to be expected, the jump in expressivity by adopting PSMs cannot come for free: the problem of computing a sound and complete projection for PSMs is in general undecidable, a fact inherited from being able to encode HMSCs. This does not defeat us, however: one of our main positive results is the definition of a very large class of PSMs, called *Tame PSMs*, that enjoys sound and complete PSPACE projection. A PSM is *tame* if it satisfies three constraints: (a) a technical refinement of the notion of the bound B for buffers, (b) that final states have no outgoing transitions, and (c) *sender-driven choice*: at each branching point, there is a single sender taking distinct actions.

Our proof works by reducing the problem to an instance of projectability of MSTs with sender-driven choice, which was proven to be decidable in PSPACE [48]. Our reduction is surprising because it produces a transformed protocol which is different from the original one: the encoded protocol language is different and involves additional participants and additional message exchanges; and yet its synthesized local specifications can be transformed back to local spec-

ifications for the original protocol. Due to the mismatch in expressivity between PSMs and global types, it is necessary that the reduction modifies the protocol semantics. Furthermore, we show the reduction preserves the complexity class, giving us a PSPACE algorithm for projectability of sender-driven PSMs.

Despite being a restriction, Tame PSMs are still much more general than global types: every sender-driven global type gives rise to a Tame PSM; moreover, every example given so far is tame. While the first two constraints (a) and (b) are not severe, the third condition (c) imposes a genuine restriction on expressivity.

In fact our main negative result is that sender-driven choice is in a sense “minimal”: we prove that projectability is undecidable for global types (the most primitive kind of PSMs) with general choice (aka “mixed choice”).

2.3 Processes and Typing

To complete the top-down toolkit, we provide a mean to check that a program correctly implements a protocol specified as a CSM. We achieve this by defining a CSM-based session type system for an expressive variant of π -calculus with session interleaving and delegation. The process calculus is adapted from [59] which represents a feature-rich modern presentation of multiparty session typing.

The type system’s main soundness argument hinges, as is standard, on a subject reduction result: if a typable program can take a step, it remains typable. From this, we derive two main safety correctness guarantees: typable programs cannot produce type mismatches (i.e. receiving a message that the process is not expecting) and terminated sessions do not leave orphan messages behind. We further prove a progress property under standard restrictions: roughly speaking, if the process contains only one session, then, if the type of the session is not final, the process can take a step (among the ones allowed by the type). Global progress in the presence of session interleaving is out of scope of this paper, but it may be attainable by adapting the (orthogonal) analysis employed in [18, 44].

In line with our decoupling goal, the guarantees of the type system are derived from the key properties of CSMs produced by projection (e.g. deadlock freedom). This makes it even compatible with the bottom-up methodology of [59] which forgoes global types and proposes to check key properties on local types directly. If a CSM satisfying the desired properties is provided to our type system, the corresponding guarantees apply to typable processes regardless of the existence of a PSM representing the protocol. This also liberates the type system completely from the choice of representation for global protocols.

Overall we obtain an expressive, decoupled and robust backend for top-down protocol development. Finally, we also show that this backend is backwards-compatible with MSTs: not only every sender-driven global type can be encoded as Tame PSM, but we also prove that, when there exists a local type that is a projection of a global type, our projection produces a CSM that can be translated back to a local type. This shows under which conditions PSMs and global types as well as CSMs and local types are equivalent, despite their structural differences.

3 Automata-based Protocol Specifications

We start our technical development by introducing a language-theoretic view of protocol specifications. We define protocols as special languages of words, and use CSMS as our local specifications of such languages. Finally, we introduce PSMs as global protocol specifications.

3.1 State Machines and Protocol Languages

Let Δ be a finite alphabet. The set of finite words over Δ is denoted by Δ^* , the set of infinite words by Δ^ω , and their union by Δ^∞ . We write ε for the empty word. For two strings $u \in \Delta^*$ and $v \in \Delta^\infty$, their concatenation is $u \cdot w$, and we say that u is a *prefix* of v , written $u \leq v$, if there is some $w \in \Delta^\infty$ such that $u \cdot w = v$; $\text{pref}(v)$ denotes all prefixes of v and is lifted to languages as expected. For a language $L \subseteq \Delta^\infty$, we distinguish between the language of finite words $L_{\text{fin}} := L \cap \Delta^*$ and the language of infinite words $L_{\text{inf}} := L \cap \Delta^\omega$.

Definition 3.1 (State machines). A state machine $A = (Q, \Delta, \delta, q_0, F)$ is a 5-tuple with a finite set of states Q , an alphabet Δ , a transition relation $\delta \subseteq Q \times (\Delta \cup \{\varepsilon\}) \times Q$, an initial state $q_0 \in Q$ from the set of states, and a set of final states F with $F \subseteq Q$. If $(q, a, q') \in \delta$, we also write $q \xrightarrow{a} q'$. A run is a finite or infinite sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \dots$, with $q_i \in Q$ and $a_i \in \Delta \cup \{\varepsilon\}$ for $i \geq 0$, such that q_0 is the initial state, and for each $i \geq 0$, it holds that $(q_i, a_i, q_{i+1}) \in \delta$. The trace of such run is the word $a_0 \cdot a_1 \dots \in \Delta^\infty$. A run is maximal if it ends in a final state or is infinite. The (core) language $\mathcal{L}(A)$ of A is the set of traces of all maximal runs. If Q is finite, we say A is a finite state machine (FSM). A state machine is dense if for every $q \xrightarrow{x} q' \in \delta$, the transition label x is ε implies that q has only one outgoing transition. A state machine is deterministic if $\forall (q, a, q') \in \delta. a \neq \varepsilon$ and $\forall (q, a, q'), (q, a, q'') \in \delta. q' = q''$. We call a dense state machine deterministic if $\forall (q, a, q'), (q, a, q'') \in \delta. q' = q''$. A state $q \in Q$ is called a sink state if it has no outgoing transitions, i.e. $\forall a \in \Delta \cup \{\varepsilon\}, q' \in Q. (q, a, q') \notin \delta$. We say a state machine is sink-final if, for every state, it is final iff it is a sink.

A language-theoretic view of protocols. Let $m \in \mathcal{V}$ be a finite set of messages and $p, q, \dots \in \mathcal{P}$ be a finite set of participants. The alphabet of p 's send and receive events is the set $\Gamma_p := \bigcup_{q \in \mathcal{P}, m \in \mathcal{V}} \{p \triangleright q!m, p \triangleleft q?m\}$. A send event $p \triangleright q!m$ records that p sent the message m to q ; a receive event $p \triangleleft q?m$ records that p received message m from q . The alphabet of all events is the set $\Gamma_{\mathcal{P}} := \bigcup_{p \in \mathcal{P}} \Gamma_p$. A paired event is a send event and its corresponding receive event: $p \rightarrow q:m := p \triangleright q!m \cdot q \triangleleft p?m$. We define the alphabet of paired events as $\Sigma_{\mathcal{P}} := \{p \rightarrow q:m \mid p, q \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$. For the remainder of the paper, we fix an arbitrary set of participants $\mathcal{P}$ and messages $\mathcal{V}$, and often write Γ for $\Gamma_{\mathcal{P}}$ and Σ for $\Sigma_{\mathcal{P}}$. Given a word, we can project it to all letters of a certain shape: for instance, $w \downarrow_{p \triangleright q!}$ is the subword of w with all of its send events where p sends any message to q . We write $\mathcal{V}(w)$ for the sequence of values in w (in the same order). In $w = w_1 \dots \in \Gamma^\infty$, a send event $w_i = p \triangleright q!m$ is *matched* by a receive event $w_j = q \triangleleft p?m$ if $i < j$

and $\mathcal{V}((w_1 \dots w_i) \Downarrow_{p \triangleright q!}) = \mathcal{V}((w_1 \dots w_j) \Downarrow_{q \triangleleft p?})$. A send event w_i is *unmatched* if there is no such receive event w_j . A language $L \subseteq \Gamma^\infty$ satisfies *feasible eventual reception* if for every finite word $w := w_1 \dots w_n \in \text{pref}(L)$ with an unmatched send event w_i , there is an extension $w \leq w' \in L$ such that w_i is matched in w' .

A sequence of send and receive events shall describe the execution of a protocol. We define when such a sequence uses channels in FIFO manner.

Definition 3.2 (FIFO Language). *A word $w \in \Gamma^\infty$ is FIFO-compliant if for each prefix w' of w , it holds that $\mathcal{V}(w' \Downarrow_{q \triangleleft p?})$ is a prefix of $\mathcal{V}(w' \Downarrow_{p \triangleright q!})$, for every $p, q \in \mathcal{P}$. We denote the set of all infinite FIFO-compliant words by FIFO_{inf} . For finite words, we require that all send events are matched. Thus, $\text{FIFO}_{\text{fin}} := \{w \mid w \text{ is FIFO-compliant and } \mathcal{V}(w \Downarrow_{p \triangleright q!}) = \mathcal{V}(w \Downarrow_{q \triangleleft p?}) \forall p, q \in \mathcal{P}\}$. We denote the (non-regular) set of all FIFO words by $\text{FIFO} = \text{FIFO}_{\text{inf}} \uplus \text{FIFO}_{\text{fin}}$. A language $L \subseteq \text{FIFO}$ is called a FIFO language.*

As the model of distributed implementation of a protocol, we adopt communicating state machines: parallel compositions of finite-control processes communicating asynchronously via point-to-point FIFO channels.

Definition 3.3 (Communicating state machines). *We call $\mathcal{A} = \{\{A_p\}_{p \in \mathcal{P}}\}$ a communicating state machine (CSM) over $\mathcal{P}$ and $\mathcal{V}$ if A_p is a finite state machine with alphabet Γ_p for every $p \in \mathcal{P}$. The semantics of a CSM $\mathcal{A}$ is the language $\mathcal{L}(\mathcal{A}) \subseteq \text{FIFO}$ whose definition is standard (see [62]). Roughly, for each pair of distinct participants $p, q \in \mathcal{P}$ there are two FIFO channels $\langle p, q \rangle, \langle q, p \rangle \in \text{Chan}$ allowing communication between the participants in the two directions. The FSM $A_p = (Q_p, \Gamma_p, \delta_p, q_{0,p}, F_p)$ describes the possible actions of participant p . A transition $(q_p, p \triangleright q!m, q'_p) \in \delta_p$ indicates that when p takes a step from q_p to q'_p , it will send a message m to q by enqueueing it in channel $\langle p, q \rangle$. Similarly, $(q_p, p \triangleleft q?m, q'_p) \in \delta_p$ prescribes the reception by p of message m from the channel $\langle q, p \rangle$. A CSM's run always starts with empty channels and each participant running its respective initial state. We denote the set of all reachable configurations (from the initial configuration) by $\text{reach}(\mathcal{A})$. A deadlock of $\{\{A_p\}_{p \in \mathcal{P}}\}$ is a reachable configuration with no outgoing transition that has at least one non-empty channel or at least one participant not in a (local) final state.*

The formal definition is given in [62]. As an example, Fig. 6 shows the three state machines constituting a CSM.

The goal of a protocol designer is to define a protocol that can be realised as a CSM. The *projectable* languages are exactly those protocols which can.

Definition 3.4 (Projections and Projectability). *A language $L \subseteq \Gamma^\infty$ is said to be projectable if there exists a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that it generates the same language (protocol fidelity), i.e., $L = \mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\})$. We say that $\{\{A_p\}_{p \in \mathcal{P}}\}$ is a projection of L .*

The asynchronous nature of CSMs makes them unable to enforce the order between certain events without explicit synchronisation. For instance, any CSM

producing a word $p \triangleright q!m \cdot r \triangleright s!m' \cdot w$ will necessarily produce also $r \triangleright s!m' \cdot p \triangleright q!m \cdot w$. Which events can be reordered is context-dependent: the events in the word $p \triangleright q!m \cdot q \triangleleft p?m$ cannot be swapped, as the receive is only possible after the send. But in $p \triangleright q!m \cdot p \triangleright q!m \cdot q \triangleleft p?m$ the last two events can be reordered. This has been formalised as equivalence relation by Majumdar et al. [51], which can be seen as an instance of Lamport's happens-before relation [46] for systems with point-to-point FIFO channels.

Definition 3.5. *The indistinguishability relation $\sim \subseteq \Gamma^* \times \Gamma^*$ is the smallest equivalence relation such that*

- (1) *If $p \neq r$, then $w \cdot p \triangleright q!m \cdot r \triangleright s!m' \cdot u \sim w \cdot r \triangleright s!m' \cdot p \triangleright q!m \cdot u$.*
- (2) *If $q \neq s$, then $w \cdot q \triangleleft p?m \cdot s \triangleleft r?m' \cdot u \sim w \cdot s \triangleleft r?m' \cdot q \triangleleft p?m \cdot u$.*
- (3) *If $p \neq s \wedge (p \neq r \vee q \neq s)$, then $w \cdot p \triangleright q!m \cdot s \triangleleft r?m' \cdot u \sim w \cdot s \triangleleft r?m' \cdot p \triangleright q!m \cdot u$.*
- (4) *If $|w \downarrow_{p \triangleright q!}| > |w \downarrow_{q \triangleleft p?}|$, then $w \cdot p \triangleright q!m \cdot q \triangleleft p?m' \cdot u \sim w \cdot q \triangleleft p?m' \cdot p \triangleright q!m \cdot u$.*

We define $u \preceq_{\sim} v$ if there is $w \in \Gamma^*$ such that $u \cdot w \sim v$. Observe that $u \sim v$ iff $u \preceq_{\sim} v$ and $v \preceq_{\sim} u$. For infinite words $u, v \in \Gamma^\omega$, we define $u \preceq_{\sim}^\omega v$ if for each finite prefix u' of u , there is a finite prefix v' of v such that $u' \preceq_{\sim} v'$. Define $u \sim v$ iff $u \preceq_{\sim}^\omega v$ and $v \preceq_{\sim}^\omega u$. We lift the equivalence relation $\sim$ on words to languages. For a language L , we define

$$\mathcal{C}(L) = \left\{ w' \mid \bigvee_{w' \in \Gamma^\omega \wedge \exists w \in \Gamma^\omega. w \in L \text{ and } w' \preceq_{\sim}^\omega w} w' \in \Gamma^* \wedge \exists w \in \Gamma^*. w \in L \text{ and } w' \sim w \right\}.$$

Lemma 3.6 ([51]). *For any CSM $\{A_p\}_{p \in \mathcal{P}}$, $\mathcal{L}(\{A_p\}_{p \in \mathcal{P}}) = \mathcal{C}(\mathcal{L}(\{A_p\}_{p \in \mathcal{P}}))$.*

Example 3.7. For finite words $\mathcal{C}(-)$ is standard. For infinite words, though, the situation is a bit counterintuitive. Let us consider $w := (p \triangleright q!m \cdot q \triangleleft p?m)^\omega$. It is easy to construct a CSM $\{A_p\}_{p \in \mathcal{P}}$, with FSMs A_p and A_q , that accepts w . CSMs do not promise any sort of fairness for infinite runs so there is an infinite run for $(p \triangleright q!m)^\omega$ where only A_p 's transitions are scheduled. This is why $\mathcal{C}(-)$ is defined using $\preceq_{\sim}^\omega$, giving $(p \triangleright q!m)^\omega \in \mathcal{C}((p \triangleright q!m \cdot q \triangleleft p?m)^\omega)$.

3.2 Protocol State Machines

We now introduce PSMs as a mean to specify protocol languages from a global, centralised perspective. The idea, shared with both global types and HMSCs, is to specify only a core subset of the admissible executions, e.g. the ones where there is a bounded delay between sends and matching receives, and obtain the full set of admissible executions by closing the core language using $\mathcal{C}(-)$.

We adapt the notion of B -bounded from [27] to formalise the idea of “bounded delay” between matching events.

Definition 3.8 (B -bounded and ΣB -bounded). *Let $B \in \mathbb{N}$ be a natural number. A FIFO-compliant word w is B -bounded, resp. ΣB -bounded, if for every prefix w' of w and participants $p, q \in \mathcal{P}$, it holds that $|w' \downarrow_{p \triangleright q!}| - |w' \downarrow_{q \triangleleft p?}| \leq B$, resp. $\sum_{p \neq q \in \mathcal{P}} (|w' \downarrow_{p \triangleright q!}| - |w' \downarrow_{q \triangleleft p?}|) \leq B$. We define the (regular) set of B -bounded FIFO words: $\text{FIFO}_B := \{w \in \text{FIFO} \mid w \text{ is } B\text{-bounded}\}$.*

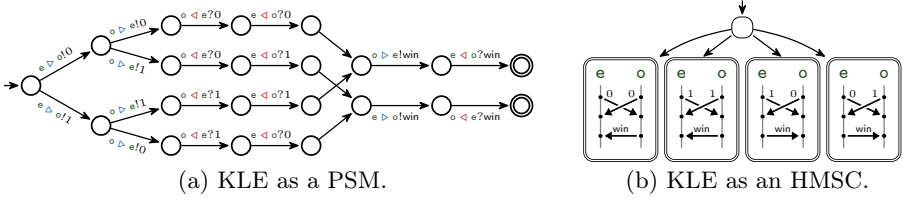


Fig. 7: Kindergarten Leader Election (KLE).

Definition 3.9 (Protocol State Machine). A dense FSM $M = (Q, \Gamma, \delta, q_0, F)$ is a B -PSM if $\mathcal{L}(M) \subseteq \text{FIFO}_B$ and $\mathcal{L}(M)$ satisfies feasible eventual reception. The semantics of M defined as $\mathcal{S}(M) := \mathcal{C}(\mathcal{L}(M))$. Moreover, M is a PSM if it is a B -PSM for some B .

By definition, PSMs specify FIFO languages; importantly, although the core language $\mathcal{L}(M)$ is B -bounded, the semantics $\mathcal{C}(\mathcal{L}(M))$ includes non- B -bounded words and will not even be regular in general. Note that, it is decidable to check if an FSM is a B -PSM.

In [62], we show that $\mathcal{C}(-)$ preserves and reflects feasible eventual reception: if $L \subseteq \Gamma^\infty$ satisfies feasible eventual reception, then $\mathcal{C}(L)$ does, and if $\mathcal{C}(L)$ satisfies feasible eventual reception, then L does. More generally, every property that is preserved by $\mathcal{C}(-)$ can be soundly checked on the core language of a PSM. If the property is also reflected by $\mathcal{C}(-)$, the property holds if and only if it holds for the core language.

Definition 3.10. A PSM M is a $\Sigma 1$ -PSM if its core language $\mathcal{L}(M)$ is $\Sigma 1$ -bounded. We may abuse notation and use $\Sigma_{\mathcal{P}}$ as alphabet for $\Sigma 1$ -PSMs.

Example 3.11 (Kindergarten Leader Election). We consider a protocol between two participants e (evens) and o (odds). It can be used to quickly settle a dispute between children (hence the name). Both children pick 0 or 1 and tell each other their pick at the same time. Child e wins if the sum is even while o wins if the sum is odd. At the end, the loser concedes by sending the message `win` to the winner. The protocol is specified as a PSM in Fig. 7a (and as an HMSC in Fig. 7b). Note that specifying this protocol requires the ability of issuing send and receive events independently. If one insisted on issuing send and matching receives together, as in global types and $\Sigma 1$ -PSMs, one of the children would be forced to reveal their hand first, undermining the purpose of the protocol.

4 Projection: From PSMs to CSMs

A CSM $\mathcal{A}$ is a projection of a PSM M , if $\mathcal{A}$ is a projection of $\mathcal{S}(M)$. In this section, we explain two main results. The first is positive: we show that sound and complete projection is decidable for Tame PSMs. The second is negative: we show that the sender-driven restriction of Tame PSMs is necessary: if we drop the restriction, projectability becomes undecidable even for sink-final $\Sigma 1$ -PSMs. The full proofs can be found in [62].

4.1 Sound and Complete Projection for Tame PSMs

The idea of the decidability result is to reduce projectability of a Tame PSM to projectability of a (different) sender-driven global type, which can then be handled using the sound and complete algorithm of [48]. Furthermore, the reduction is such that a projection of the original PSM can be read off a projection of the global type. Before sketching the idea behind the reduction, we define Tame PSMs formally. Tame PSMs satisfy three conditions: they are sink-final, sender-driven, and satisfy some more fine-grained bounds on the message queues.

Definition 4.1 (Choice restrictions for PSMs). *Let $M = (Q, \Gamma, \delta, q_0, F)$ be a PSM. The PSM M satisfies sender-driven choice if there is a function $\lambda: Q \rightarrow \mathcal{P}$ such that for all states q, q' such that $q \xrightarrow{x} q'$ with $x \in \Gamma$, it holds that $\lambda(q)$ is the sender for x , i.e., $x = \lambda(q) \triangleright \_!$. In addition, we say M is directed if for every state q , there is also a dedicated receiver p , i.e., all transition labels from q are of the form $\lambda(q) \triangleright p! \_$. Last, if there is no dedicated sender but all transitions are still distinct, i.e. M is deterministic, we say that it satisfies mixed choice.*

Definition 4.2 (Channel bounds for PSMs). *We define channel bounds as a partial function $\beta: \text{Chan} \rightarrow \mathbb{N}$ from channels to natural numbers, where $\text{dom}(\beta)$ denotes the domain of β . Given a PSM M , we say that M respects β if the following holds for every $\langle p, q \rangle \in \text{Chan}$:*

- *If $\langle p, q \rangle \notin \text{dom}(\beta)$, then every message from p to q is immediately followed by a receive: for every state q and transition from q to q' labelled with $p \triangleright q!m$, it holds that there is a single transition from q' and it is labelled with $q \triangleleft p?m$.*
- *If $\langle p, q \rangle \in \text{dom}(\beta)$, then $w \Downarrow_{p \triangleright q! \cdot q \triangleleft p?}$ is $\beta(\langle p, q \rangle)$ -bounded for every $w \in \mathcal{S}(M)$.*

A PSM that respects β with $\beta = \emptyset$ is a PSM which only uses paired events, just like global types do. Thus checking the condition with $\beta = \emptyset$ is a trivial syntactic check. For general PSMs, it is possible to generate valid channel bounds with a sound algorithm we propose in [62]. We conjecture the algorithm to be also complete, i.e. to always output some bounds if they exist.

Definition 4.3 (Tame PSMs). *A Tame PSM is a pair (M, β) where the PSM M is sender-driven, sink-final, and respects the channel bounds β .*

We can now sketch the idea behind the reduction. Fundamentally, the gap in expressivity between Tame PSMs and sender-driven global types is that in PSMs sends and matching receives do not need to appear one right after the other. One can observe, however, that one could replicate the same asynchrony of some trace $p \triangleright q!m \cdots q \triangleleft p?m$ by introducing an intermediary participant (p, q) that is always ready to forward messages from p to q , leading to a trace $p \rightarrow (p, q):m \cdots (p, q) \rightarrow q:m$ where the sends and matching receives between participants and the intermediaries are now immediately adjacent. The channel bounds β tell us exactly for which channels we need to introduce intermediaries; moreover the bound on the buffers induced by β makes sure that these intermediaries will not introduce any spurious dependency in the executions. To consolidate the idea, we show how it applies to our KLE example.

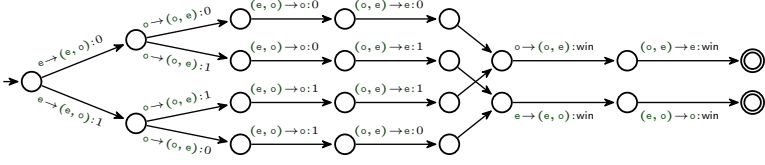


Fig. 8: Kindergarten Leader Election after the Channel-participant Encoding.

Example 4.4 (Revisiting the KLE protocol). In Example 3.11, we introduced the Kindergarten Leader Election protocol, whose communication pattern cannot be represented as a $\Sigma 1$ -PSM/global type: both children need to commit to the number they send before they receive the other’s message. Its PSM (Fig. 7) is however tame: it is sink-final, sender-driven and respects $\beta(e, o) = \beta(o, e) = 1$. The “intermediary forwarders” idea applied to the protocol results in a protocol where some teachers (the intermediaries) will act as depositories for the initial choices of the two children. After committing their choice, each child is allowed to learn from the teacher the choice of the other child. The resulting PSM is given in Fig. 8. The names of the additional participants indicate the direction of communication: (e, o) forwards messages from e to o . Obviously, this encoding does not specify the same protocol. Still, our construction shows that one can obtain a projection of the original protocol from a projection of the modified one, by appropriately removing the forwarding actions of the teachers.

The example illustrates the simple case where $\beta(-) \leq 1$; in the more general case, the reduction is more involved and requires more intermediaries.

The workflow of our encoding is visualised in Fig. 9. Given a PSM M , one first computes its encoding $\text{enc}_{\text{PSM}}(M)$. Second, one synthesizes a projection $\{\{A_p\}\}_{p \in \mathcal{P}}$ of the encoded protocol using results from [48]. Third, one decodes to obtain a projection $\{\{\text{dec}_{\text{FSM}}(A_p)\}\}_{p \in \mathcal{P}}$ of M .

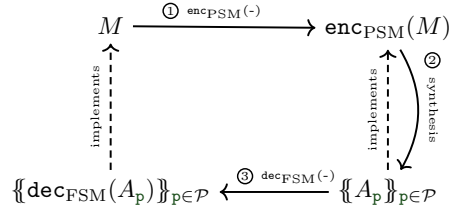


Fig. 9: Workflow of encoding.

Theorem 4.5. *Checking projectability of Tame PSMs is in PSPACE. One can also synthesize a projection in PSPACE.*

4.2 Mixed Choice yields Undecidable Projectability

Now, we show that the sender-driven choice restriction for Tame PSMs is necessary for projectability to be decidable. General PSMs inherit undecidability of projectability from HMSCs, which in turn was proven by Lohrey [50, Thm. 3.4]. Given our positive result for Tame PSMs, the proof for undecidability ought to break in the presence of sender-driven choice. The original proof goes through several (often implicitly given or omitted) automata-based transformations and does not give any insights about where and how the transformations break under the assumption of sender-driven choice.

Theorem 4.6. *The projectability problem for sink-final mixed-choice $\Sigma 1$ -PSMs is undecidable.*

We reduce the membership problem for Turing Machines to checking projectability of a sink-final mixed-choice $\Sigma 1$ -PSM with five participants. Initially, there is a branching which only two participants are involved in and learn about. Subsequently, all participants communicate Turing machine computations in the form of configurations in both branches. If the (projected) language of one of the other participants is not the same for both branches, the PSM cannot be implementable because they do not know which branch to comply with and easily deadlock. We also show that the reverse is the case. Hence, we specify a language for each branch and make both coincide if and only if the Turing Machine has no accepting computation, which is the case if and only if the PSM is projectable.

The full proof is in [62]. We adopt the proof strategy of Lohrey to PSMs and make every transformation explicit and carefully check which structural properties the transformations preserve, yielding a stronger undecidability result concerning the most rudimentary of PSMs: $\Sigma 1$ -PSMs.

5 Typing Programs against CSMs

We now overview the key ideas behind AMP’s type system. The formal details and full proofs can be found in [62]. To define programs, we take inspiration from the process calculus with session interleaving and delegation of [59]. The syntax of AMP’s programs is reproduced in Fig. 10. The processes P represent the static program text. As is standard, $\mathbf{0}$ is the terminated process, $\parallel$ denotes parallel composition, $Q[\vec{c}]$ denotes a sequential process running the code defined by a finite set of definitions $\mathcal{D}$. The prefixes $\oplus_{i \in I} c[\mathbf{q}_i]!l_i \langle c_i \rangle$ and $\&_{i \in I} c[\mathbf{q}_i]?l_i(y_i)$ denote internal and external choice respectively, with a non-empty finite set of indices I . The endpoint of participant $\mathbf{p}$ of a channel between $\mathbf{p}$ and $\mathbf{q}$ in a session s , is denoted by $s[\mathbf{p}][\mathbf{q}]$; $\mathbf{p}$ can send a label l and some payload p to $\mathbf{q}$ in session s by $s[\mathbf{p}][\mathbf{q}]!l \langle p \rangle$, the dual reception is denoted by $s[\mathbf{q}][\mathbf{p}]?l \langle x \rangle$ (which binds the payload to x). To model delegation, a process must be able to send to another the capability to act as participant $\mathbf{p}$ in session s , denoted $s[\mathbf{p}]$; the receiving process will bind such capability to a variable x and use it to form endpoints $x[\mathbf{q}]$; we thus have in general send/receive actions on $c[\mathbf{q}]$ where c can be a variable or some $s[\mathbf{p}]$.

The process $(\mathbf{vs} : \mathcal{A}) P$ denotes the creation of a new bound session s used in P . The session is annotated with a (computationally irrelevant) CSM $\mathcal{A}$, taking the place of what is often a global type. So far, we treated messages in CSMs very abstractly as elements of a finite alphabet. In processes, messages are more structured: they have a label (from a finite set) and a payload (of some type). The messages used by the CSM will thus be pairs $l(t)$ of a label l and a payload type t , with the convention that if, from a state q , there are two outgoing transitions with the same sender, receiver and label, they will agree on the type.

In applications, the payload can be of any base type (e.g. integers, strings), or be a channel capability $s[\mathbf{p}]$ (for delegation). Since supporting base types is a

$$\begin{aligned}
c &::= x \mid s[p] \\
P &::= \mathbf{0} \mid P_1 \parallel P_2 \mid (\mathbf{vs} : \mathcal{A}) P \mid \bigoplus_{i \in I} c[\mathbf{q}_i]!l_i\langle c_i \rangle . P_i \mid \bigotimes_{i \in I} c[\mathbf{q}_i]?l_i(y_i) . P_i \mid \mathbf{Q}[\vec{c}] \\
R &::= \mathbf{0} \mid R_1 \parallel R_2 \mid (\mathbf{vs} : \mathcal{A}) R \mid \bigoplus_{i \in I} c[\mathbf{q}_i]!l_i\langle c_i \rangle . P_i \mid \bigotimes_{i \in I} c[\mathbf{q}_i]?l_i(y_i) . P_i \mid \mathbf{Q}[\vec{c}] \\
&\quad \mid s \blacktriangleright \sigma \mid \mathbf{err} \\
\mathcal{D} &::= (\mathbf{Q}[\vec{x}] = \bigoplus_{i \in I} c[\mathbf{q}_i]!l_i\langle c_i \rangle . P_i); \mathcal{D} \mid (\mathbf{Q}[\vec{x}] = \bigotimes_{i \in I} c[\mathbf{q}_i]?l_i(y_i) . P_i); \mathcal{D} \mid \varepsilon
\end{aligned}$$

Fig. 10: Syntax of AMP’s π -calculus.

simple exercise, we follow [59] and focus on the harder case of channel capabilities as payloads. When using a CSM $\mathcal{A} = \{\{A_p\}_{p \in \mathcal{P}}\}$ as a protocol specification for a session s , it is natural to consider the (control) states Q_p of A_p to be the local types that can be associated to $s[p]$. Therefore, in our setting we will consider the set L of the states of any $\mathcal{A}$ annotating the process, as the possible payload types. For simplicity, we assume all CSMs use disjoint sets of states, so that we can unambiguously refer to the transitions from any state q by $\delta(q)$.

In particular, if the protocol specified by $\mathcal{A}$ can delegate channels of a session following some CSM $\mathcal{B}$, then the message alphabet of $\mathcal{A}$ will include states of $\mathcal{B}$. When the CSMs are obtained through projection, it is natural to first obtain $\mathcal{B}$ so we can refer to its states in writing the PSM that projects to $\mathcal{A}$. We thus assume there is an acyclic “delegation partial order” between the CSMs of a process: $\mathcal{B} < \mathcal{A}$ means that $\mathcal{A}$ can use the states of $\mathcal{B}$ in its messages.

The semantics of the calculus is defined on runtime configurations R (defined in Fig. 10), which are processes which additionally contain message queues $s \blacktriangleright \sigma$ for each active session s . Here σ is a map from pairs of participants to sequences of messages. The reduction semantics is standard (cf. [62]). The only reduction rules we highlight here are the ones leading to an error configuration:

$$\frac{\forall i \in I. \sigma(\mathbf{q}_i, \mathbf{p}) = l(\_)\dots \wedge l_i \neq l}{\bigotimes_{i \in I} s[p][\mathbf{q}_i]?l_i(y_i) . P_i \parallel s \blacktriangleright \sigma \longrightarrow \mathbf{err}} \quad \frac{\sigma(\mathbf{p}, \mathbf{q}) \neq \varepsilon \text{ for some } \mathbf{p}, \mathbf{q}}{(\mathbf{vs} : \mathcal{A}) s \blacktriangleright \sigma \longrightarrow \mathbf{err}}$$

The first rule models *unsafe communication*: a process is stuck because all the queues it is waiting to receive from are not empty, but the labels of the first messages do not match any of the cases the process is expecting. The second rule models *orphan messages*: a session where all participants terminated but that has still non-empty message queues. The safety guarantees of our type system will rule out both cases. Note that [58, 59] focuses on communication safety. In addition, they consider S-deadlock freedom, which implies no orphan messages, but is an undecidable property that needs to be checked and is not necessarily transferred to processes by the type system: the property only holds if there is only one session, in which case much stronger conditions transfer. In our setting, deadlock freedom is transferred throughout by projection and the type system, yielding no orphan messages.

Figure 11 shows the crucial rules of AMP’s type system. The typing judgement $\Theta \vdash \Lambda \vdash P$ uses a process P , a typing context Θ for the types of the parameters $\vec{c}$ of sequential processes $\mathbf{Q}[\vec{c}]$ (the definitions of which are typed separately

$$\begin{array}{c}
\frac{}{\Theta \vdash \emptyset \vdash \mathbf{0}} \text{PT-}\mathbf{0} \qquad \frac{\Theta \vdash \Lambda \vdash P \quad \text{end}(q)}{\Theta \vdash c : q, \Lambda \vdash P} \text{PT-END} \\
\\
\frac{\delta(q) = \{(p \triangleleft q_i ? l_i(p_i), q_i) \mid i \in I\} \quad \forall i \in I. \Theta \vdash \Lambda, c : q_i, y_i : p_i \vdash P_i}{\Theta \vdash \Lambda, c : q \vdash \&_{i \in I} c[q_i] ? l_i(y_i) . P_i} \text{PT-}\& \\
\\
\frac{\delta(q) \supseteq \{(p \triangleright q_i ! l_i(p_i), q_i) \mid i \in I\} \quad \forall i \in I. \Theta \vdash \Lambda, c : q_i, \{c_j : p_j\}_{j \in I \setminus \{i\}} \vdash P_i}{\Theta \vdash \Lambda, c : q, \{c_i : p_i\}_{i \in I} \vdash \oplus_{i \in I} c[q_i] ! l_i(c_i) . P_i} \text{PT-}\oplus \\
\\
\frac{\Lambda_s = \{s[p] : \text{init}(\mathcal{A}_p)\}_{p \in \mathcal{P}_A} \quad \Theta \vdash \Lambda, \Lambda_s \vdash P}{\Theta \vdash \Lambda \vdash (vs : \mathcal{A}) P} \text{PT-v}
\end{array}$$

Fig. 11: Typing rules for processes; $\text{init}(-)$ denotes a CSM's initial state.

against Θ); a typing context Λ associating the variables x and the channel capabilities $s[p]$ occurring free in P , with some CSM state $q \in L$. Rule PT- $\mathbf{0}$ says that a terminated process is typable in the environment with no capabilities. Rule PT-END permits to discard the capabilities that have terminated: $\text{end}(q)$ holds for final states with no outgoing receive transition. Rules PT- $\&$ and PT- $\oplus$ deal with communication. Assume $c = s[p]$. According to PT- $\&$, to receive a message as participant p in session c , we look for the type q of c in the typing context and check the CSM transitions $\delta(q)$ are all receives $\{(p \triangleleft q_i ? l_i(p_i), q_i) \mid i \in I\}$. Then the process needs to be able to receive any branch i , resulting in the continuation P_i which is typed in the context extended with the corresponding payload type $y_i : p_i$, and with the type of c changed to q_i . According to PT- $\oplus$, to send, as c with type q , a message non-deterministically picked from a number of branches $i \in I$, we have to make sure q allows each branch, including matching the types of the payloads. Then each branch i continues as P_i which is typed in a context where c has type q_i and we lost ownership of the payload c_i . Finally, PT-v types a new session s used by P , by adding to the context a new binding $s[p] : q_p$ for each participant p of the CSM $\mathcal{A}$ annotating the session, with q_p being the initial state of p in $\mathcal{A}$.

The first correctness criterion for the type system is to prove subject reduction: if a process is typable, then every configuration reachable from it will be typable. Thus, to state subject reduction, we need to define when a runtime configuration is typable. For this purpose, we define a second judgement $\Theta \vdash \Lambda \vdash \Omega \vdash R$ that includes a third typing context Ω used to type session message queues: associating to each channel $s[p][q]$ a sequence of message types $l(q)$ (label and payload type). The key to make typing of runtime configurations an inductive invariant, is the following rule:

$$\frac{\begin{array}{c} (\vec{q}, \xi) \in \text{reach}(\mathcal{A}) \quad \Lambda_{\vec{q}}^s = \{s[p] : \vec{q}_p\}_{p \in \mathcal{P}_A} \\ \Omega_{\xi}^s = \{s[p][q] :: \xi(p, q)\}_{p, q \in \mathcal{P}_A} \quad \Theta \vdash \Lambda, \Lambda_{\vec{q}}^s \vdash \Omega, \Omega_{\xi}^s \vdash R \end{array}}{\Theta \vdash \Lambda \vdash \Omega \vdash (vs : \mathcal{A}) R} \text{RT-v}$$

The main difference between RT-**v** and PT-**v** is that the typing context is not populated with capabilities associated to initial states; instead the prover can pick any CSM configuration $(\vec{q}, \xi)$ —where $\vec{q}$ collects the local state of each participant, and ξ the contents of the message queues— that is reachable from the initial configuration of $\mathcal{A}$. The states and the queues are used to initialise the typing context to type the process R using the restricted session.

In what follows we assume the definitions $\mathcal{D}$ can be typed according to Θ . We say a process/runtime configuration is *well-annotated* if every CSM appearing in it is (1) deadlock-free, and (2) satisfies feasible eventual reception. Here, *annotated* indicates that the CSMs have no computational meaning but *well* shows the need for certain guarantees, which our type system can preserve. Note that a process is automatically well-annotated if the CSMs are obtained via projection.

Theorem 5.1 (Subject Reduction). *Given a well-annotated R , if $\Theta \vdash \emptyset \vdash \emptyset \vdash R$ and $R \longrightarrow R'$, then $\Theta \vdash \emptyset \vdash \emptyset \vdash R'$.*

Corollary 5.2 (Type Safety). *For a well-annotated R , if $\Theta \vdash \emptyset \vdash \emptyset \vdash R$ and $R \longrightarrow^* R'$, then R' does not contain **err**.*

For progress, the situation is more delicate: just like in [59] and most other MST systems, allowing session interleaving may introduce inter-session dependencies that are not modelled in the global protocol (which only pertains intra-session dependencies). We thus prove progress under these assumptions: (i) there is only one session running, and (ii) that each of its participants is implemented by a distinct process, and (iii) the CSM annotating it is sink-final. To encode these extra restrictions, we define a “Session Fidelity” variant of the typing judgement $\Theta \vdash \Lambda \vdash \Omega \vdash_{SF} R$ which uses a subset of the rules of $\vdash$ to enforce the restrictions above. Let $\Lambda_{\vec{q}}^s$ and Ω_{ξ}^s be defined as in the premises of RT-**v**.

Theorem 5.3 (Progress). *Let $(\vec{q}, \xi)$ be a configuration of a sink-final, deadlock-free CSM $\mathcal{A}$ satisfying feasible eventual reception. If $\Theta \vdash \Lambda_{\vec{q}}^s \vdash \Omega_{\xi}^s \vdash_{SF} R$, and $(\vec{q}, \xi)$ can take a step, then there exist some R' and $(\vec{q}', \xi')$, such that $R \longrightarrow R'$, $(\vec{q}, \xi) \longrightarrow (\vec{q}', \xi')$, and $\Theta \vdash \Lambda_{\vec{q}'}^s \vdash \Omega_{\xi'}^s \vdash_{SF} R'$.*

Progress hinges on deadlock freedom of the CSM. In general, any (language) property of a PSM that is preserved and reflected by $\mathcal{C}(-)$ holds for its projection. However, as for progress, it is not necessarily easy to make the type system enforce the preservation of these properties at the global process level and requires careful treatment. [18] demonstrated how Kobayashi-style techniques [44] that can be used to show progress in the presence of session interleaving. We conjecture a similar system can be added on top of AMP’s type system.

6 Applications of AMP to MST Frameworks

Standard (expression-based) global types from MST frameworks can be seen as restricted special cases of PSMs. What is gained from using AMP for global types seen as PSMs? Is anything lost in doing so? In this section, we evaluate AMP as

a backend for projection/typing of standard global types. The key consequences of our results are:

- (a) Every sender-driven global type is a tame sink-final $\Sigma 1$ -PSM.
- (b) Tame sink-final $\Sigma 1$ -PSMs can be represented as a sender-driven global type.
- (c) Every collection of (expression-based) local types $\{\{L_p\}_{p \in \mathcal{P}}\}$ can be expressed as a CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ and vice versa.
- (d) AMP’s projection is deadlock-free by construction, but MSTs typically insist on freedom of a stricter notion of deadlock which we call *soft deadlock*. We show AMP’s projection can also be set to ensure soft deadlock freedom, without losing completeness.

These results help us settle two open questions:

- Expression-based global/local types are equi-expressive with respect to state-machine-based global/local specifications.
- Allowing mixed-choice in global types makes projectability undecidable.

Here, we give an overview while [62] presents the results in detail.

Global and local types. In most MST frameworks, protocols are specified using expression-based global types (G), which get projected to expression-based local types (L). Their syntax is specified as follows:

$$\begin{aligned}
 G &::= \mathbf{0} \mid \sum_{i \in I} \mathbf{p}_i \rightarrow \mathbf{q}_i : m_i . G_i \mid \mu X . G \mid X \quad (\text{global types}) \\
 L &::= \mathbf{0} \mid \bigoplus_{i \in I} \mathbf{q}_i ! m_i . L_i \mid \&_{i \in I} \mathbf{q}_i ? m_i . L_i \mid \mu X . L \mid X \quad (\text{local types})
 \end{aligned}$$

where $\mathbf{0}$ explicitly denotes termination and μX binds a recursion variable X . The remaining operators specify how messages are exchanged: for local types, sending and receiving are separate actions, while for global types they are specified in a single paired event. Typically only *deterministic* global types are considered, i.e. where every $\sum_{i \in I} \mathbf{p}_i \rightarrow \mathbf{q}_i : m_i . G_i$ has no $i \neq j$ with $\mathbf{p}_i \rightarrow \mathbf{q}_i : m_i = \mathbf{p}_j \rightarrow \mathbf{q}_j : m_j$. The choice restrictions we discussed, can be imposed on global types, e.g. sender-driven choice requires that, for $\sum_{i \in I} \mathbf{p}_i \rightarrow \mathbf{q}_i : m_i . G_i$, for all $i, j \in I$, $\mathbf{p}_i = \mathbf{p}_j$.

The standard semantics of global types has been given as a transition system, or as sets of traces. In both cases, the semantics allows reordering of events that are not causally related, e.g. $\mathbf{p} \rightarrow \mathbf{q} : 1 . \mathbf{r} \rightarrow \mathbf{s} : 2 . \mathbf{0}$ allows $\mathbf{r}$ and $\mathbf{s}$ to communicate before $\mathbf{p}$ and $\mathbf{q}$. This is formalised, in the presentation of [48, 51, 60] (which we adopt here) as defining the semantics of a global type to be a set of traces closed under the indistinguishability relation $\sim$. With this view, it is immediate to represent any global type as a PSM. Given the restricted format of global types, the PSMs corresponding to translations of global types (like the one in Fig. 4) are $\Sigma 1$ -PSMs with a specific shape: they are tree-like, sink-final and recursion only happens at leaves and to ancestors [60]. On the face of it, it is unclear whether every $\Sigma 1$ -PSM can be modelled as global type.

Theorem 6.1. *For every sink-final $\Sigma 1$ -PSM M , there is a global type $\text{GAut}(M)$ with the same core language (and hence the same semantics). If M is non-deterministic (mixed-choice, sender-driven, or directed, resp.), then $\text{GAut}(M)$ is non-deterministic (mixed-choice, sender-driven, or directed, resp.).*

The main idea of Theorem 6.1 is that one can see a global type as a special regular expression, and thus we can adapt techniques like Arden's lemma and Brzozowski derivatives to the case of PSMs. The key difficulty in the proof lays in showing the branching conditions are preserved: the standard automata transformations change the branching structure, and we need to produce new variants that do.

Similarly, local types can be directly read as the FSMs of a CSM. We can also provide an inverse transformation (preserving branching).

Theorem 6.2. *Let A_p be a sink-final FSM over Γ_p without mixed-choice states for a participant p . One can construct a local type L_p for p with $\mathcal{L}(L_p) = \mathcal{L}(A_p)$.*

Deadlocks and protocol termination. In MSTs, local types can only terminate with a $\mathbf{0}$, which signals at the same time that it is valid to stop the protocol, and that there is no further potential action. This implies that for a global type to be projectable into local types, all the participants need to know unambiguously when the protocol terminated globally. In contrast, using CSMs, it is possible to mark as final a state with outgoing transitions. Consider for instance the (directed) global type $G := (p \rightarrow q : m_1 . p \rightarrow r : m_1 . \mathbf{0}) + (p \rightarrow q : m_2 . \mathbf{0})$. AMP's projection would produce the FSM $\begin{array}{c} \textcircled{\bullet} \xrightarrow{r \triangleleft p?m_1} \textcircled{\bullet} \end{array}$ as the projection for r . It contains a non-sink final state: r is not informed of which branch was taken and needs to be prepared to terminate, or receive one more message.

AMP's projection produces deadlock-free CSMs, where deadlocks are defined as configurations which cannot take a step, but their queues are not empty or some participant is in a non-final state. Projections to local types ensure the absence of another type of deadlock: a *soft deadlock*, i.e. a configuration that is a deadlock, or that cannot take a step but where some participant is in a *non-sink* state. Is the possibility of soft deadlocks desirable? We argue that this depends on the domain of application: in distributed computing, it would be fine if a server kept listening for incoming requests while, in embedded computing, it can be key that all participants eventually stop. We can show that it is possible to use AMP in both scenarios, without giving up on completeness.

Definition 6.3 (Strong Projectability). *A language $L \subseteq \Gamma^\omega$ is said to be strongly projectable if there exists a CSM $\{\{B_p\}_{p \in \mathcal{P}}\}$ such that $\{\{B_p\}_{p \in \mathcal{P}}\}$ is free from soft deadlocks (soft deadlock freedom), and L is the language of $\{\{B_p\}_{p \in \mathcal{P}}\}$ (protocol fidelity). We say that $\{\{B_p\}_{p \in \mathcal{P}}\}$ is a strong projection of L .*

Theorem 6.4. *Let G be a projectable global type. Then, the subset construction $\mathcal{C}(G, p)$ [48, Def. 5.4] is sink-final for every participant p if and only if there is a CSM that is a strong projection of G and this CSM satisfies feasible eventual reception or every of its state machines is deterministic.*

If we aim for a strong projection of a projectable global type, we construct the global type's subset construction and check if it is sink-final. If it is not, there is no strong projection of it. If this is undesirable, the protocol needs redesigning. Theorem 6.2 can yield local types and $\text{LAut}(L)$ is the FSM for a local type L .

Undecidability for mixed-choice. Finally, these results together with our results from Section 4.2, can settle the open question of whether we can project mixed-choice global types algorithmically.

Corollary 6.5. *Both the projectability problem and the strong projectability problem for mixed-choice global types are undecidable.*

7 Related Work

Multiparty session types. Inspired by linear logic [33], Honda [36] proposed binary sessions types for sessions with two participants. Multiparty session types [37] extended the idea to multiple participants. Deniérou and Yoshida [23] were the first to extensively explore the relation between CSMs and local types, but their projection is not complete and only supports directed choice; moreover the approach was found to be somewhat flawed [59]. MSTs have been incorporated into a number of programming languages [3, 17, 42, 47, 49, 54, 57]. They have also been applied to various other domains like operating systems [25], web services [71], distributed algorithms [45], timed systems [8], cyber-physical systems [52], and smart contracts [22]. A number of works are devoted to mechanising MST meta-theory [14, 40, 41, 64]. Our results could potentially extend the expressivity of the types involved in these applications.

MST projectability/projection. Via a reduction to globally-cooperative HMSCs, Stutz [60] proved MST projectability to be decidable for the class of global types that can —but do not have to— terminate (called **0**-reachable). Li et al. [48] provided a direct MST projection algorithm that is complete for sender-driven global types, providing a PSPACE upper bound. Our results use a reduction to these later developments. The global specifications in [48] can be shown to be special cases of Tame $\Sigma 1$ -PSMs so our results strictly expand the reach of their results. For example, the protocols of Figs. 5 and 7 are all tame, yet out of the reach of both works. We also clarify the discrepancy between the notion of deadlock in global types and in [48] (cf. Section 6). Finally, [48, 60] do not have a type system, providing no way to link properties of projections with the ones of processes. Preliminary versions of some of our results appeared in the PhD thesis of the first author [61].

The almost totality of asynchronous MST works can only handle directed choice. An exception is [12], where unrestricted global types are considered (without a type system). They propose an incomplete projection algorithm that is correct with respect to a different notion of correct projection than the standard one we adopt and generalise. We refer to [51] for a survey on choice restrictions.

Hu and Yoshida [38] propose a scheme with global types and an incomplete projection, where the global types are not safe by construction and the restrictions on choice only appear at the local types. The safety of global types is ensured by a combination of model-checking with message buffers of size 1, and syntactic restrictions that ensure that any unsafety that might arise, will be visible in the 1-bounded executions. For PSMs satisfying the syntactic restrictions, the same approach could be applied. The types of [38] also include

connect/disconnect actions, which can be emulated in AMP by excluding deadlocks (but not soft deadlocks) and using non-sink final states.

Choreography automata and languages. Choreography Automata [4] are syntactically similar to $\Sigma 1$ -PSMs, but do not employ any closure operation, requiring the user to specify all the allowed interleavings, and preventing finite state representations for many common communication patterns. In addition, Majumdar et al. [51] showed that their conditions for projectability are flawed for the asynchronous case (fixes for the synchronous case appeared in [31, p. 8]). Barbanera et al. [5] applies a language-theoretic approach to a limited class of *synchronous* choreographies (with no claim of completeness of projection).

Bottom-up MSTs. A number of MST-based works deviate from the top-down approach. For instance, [59] proposes a type system that only requires local types and not a global type. The typing ensures some operational correspondence between local types and processes, making it possible to model check local types to determine properties of the program. Their local types in the asynchronous setting are Turing-powerful, and therefore model checking is of limited use. By virtue of the decoupling achieved by our type system, AMP can be used in a bottom-up way too: safety of the CSM used to type a process, implies safety of the process, regardless of whether the CSM is obtained by projection or just given. Dagnino et al. [21] and Castellani et al. [13] also use a bottom-up strategy by reconstructing a so-called *deconfined* global type from the parallel composition of local programs of a single session. Deconfined global types are not automatically safe, and checking their safety is shown to be undecidable.

Extensions for MSTs. A number of extensions for MSTs have been considered (see [7] for a survey), including: parametrisation [16, 24], dependent types [24, 66, 67], graduality [39], fault-tolerance [6, 70]. Context-free session types [43, 65] specify binary sessions that are not representable with finite-control. It would be interesting to consider projection for PSMs generated by pushdown automata.

Distributed synthesis. In automata theory, distributed synthesis seeks a way to transform a sequential specification into an equivalent distributed implementation, which is close in spirit with the idea of extracting local types from global types. One of the few positive results in this area is Zielonka’s theorem [72], which shows that every regular trace language can be recognised by a so-called “asynchronous automaton”. Despite their name, asynchronous automata can be seen as a parallel composition of participants interacting through *synchronous* actions. In contrast, PSMs and CSMs represent non-regular FIFO languages, giving rise to a harder challenge.

High-level message sequence charts. HMSCs were defined in an industry standard [69], inspiring extensive academic research [26, 28, 29, 53, 56]. Projectability has been studied for HMSCs under the name “safe realisability” [2, 30, 50], and was shown to be undecidable in general [50]. Several restrictions of HMSCs have been proposed to make projectability decidable. For a detailed survey, we refer to [60]. Compared to PSMs, HMSCs only model finite runs; their PSM encoding equips them with an infinite run semantics. With our developments, it is

fairly straightforward to obtain a projection operation for sender-driven, sink-final HMSCs that respect some channel bounds. This class is incomparable to any of the decidable HMSC classes proposed in the literature. Since our type system only depends on CSMs, regardless of how they are obtained, AMP can type check a program against a projectable HMSC.

Choreographic programming. Choreographic programming [19, 32, 35] adopts the top-down approach even more radically than MSTs. In choreographic programming, the endpoint projection (EPP) aims at synthesizing a fully-featured program implementation directly from the global specification. As a result, the global specification describes the local computation alongside the communication structure (requiring infinite-state formalisms). In choreographies, one typically works with non-finite-control-state global specifications, so the hopes for a complete and decidable EPP are slim, justifying giving up on completeness. By only considering local computation in processes, the MST/AMP approach avoids this issue. Nevertheless, our results could still be useful for EPP when applied to the pure communication structure of choreographies. Notably, our method can project examples that cannot be projected using EPPs from the literature. Consider the choreography **if** $p \star$ **then** $(p \rightarrow q: \_ . q \rightarrow s: \_)$ **else** $(p \rightarrow s: \_ . s \rightarrow q: \_)$, where message payloads are irrelevant and hence omitted and $p \star$ denotes non-deterministic choice by p . The example is syntactically valid in [20] and can be easily encoded as a global type with sender-driven choice. However, their EPP would be undefined for q and s : it uses the merge from [11], which can only merge same sender receives. Our results would instead produce the desired projection.

Communicating state machines. CSMs are the canonical automata model for distributed systems. They have been studied in the context of model checking projections and do not apply a top-down methodology. The verification problem is undecidable in general since CSMs are Turing-powerful [10]. Several strategies to yield decidable classes have been proposed: assuming channels are lossy [1], restricting the communication topology [55, 68], or only allowing half-duplex communication for two participants [15]. The concept of existential boundedness [27] was initially defined for CSMs and yields decidability of control state reachability. The same holds for synchronisability [9, 34], which, intuitively, requires that every execution can be re-ordered (up to $\sim$) into phases of sends and receives such that messages can only be received in the same phase. Global types can only express 1-synchronisable and half-duplex communication [63].

Acknowledgments. The authors thank Anca Muscholl and Jorge A. Pérez for their discussions. This work was partially supported by the Luxembourg National Research Fund (FNR) under the grant agreement C22/IS/17238244/AVVA and by the ERC Consolidator Grant for the project “PERSIST” under the EU’s Horizon 2020 research and innovation programme (grant No. 101003349).

Bibliography

- [1] Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings, Lecture Notes in Computer Science, vol. 1427, pp. 305–318, Springer (1998), <https://doi.org/10.1007/BFb0028754>
- [2] Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. *Theor. Comput. Sci.* **331**(1), 97–114 (2005), <https://doi.org/10.1016/j.tcs.2004.09.034>
- [3] Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. *Found. Trends Program. Lang.* **3**(2-3), 95–230 (2016), <https://doi.org/10.1561/25000000031>
- [4] Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings, Lecture Notes in Computer Science, vol. 12134, pp. 86–106, Springer (2020), https://doi.org/10.1007/978-3-030-50029-0_6
- [5] Barbanera, F., Lanese, I., Tuosto, E.: Formal choreographic languages. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings, Lecture Notes in Computer Science, vol. 13271, pp. 121–139, Springer (2022), https://doi.org/10.1007/978-3-031-08143-9_8
- [6] Barwell, A.D., Scalas, A., Yoshida, N., Zhou, F.: Generalised multiparty session types with crash-stop failures. In: Klin, B., Lasota, S., Muscholl, A. (eds.) 33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland, LIPIcs, vol. 243, pp. 35:1–35:25, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), <https://doi.org/10.4230/LIPIcs.CONCUR.2022.35>
- [7] Bejleri, A., Domnori, E., Viering, M., Eugster, P., Mezini, M.: Comprehensive multiparty session types. *Art Sci. Eng. Program.* **3**(3), 6 (2019), <https://doi.org/10.22152/programming-journal.org/2019/3/6>
- [8] Bocchi, L., Murgia, M., Vasconcelos, V.T., Yoshida, N.: Asynchronous timed session types - from duality to time-sensitive processes. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on

- Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11423, pp. 583–610, Springer (2019), https://doi.org/10.1007/978-3-030-17184-1_21
- [9] Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part II, Lecture Notes in Computer Science, vol. 10982, pp. 372–391, Springer (2018), https://doi.org/10.1007/978-3-319-96142-2_23
 - [10] Brand, D., Zafiropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983), <https://doi.org/10.1145/322374.322380>
 - [11] Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* **34**(2), 8:1–8:78 (2012), <https://doi.org/10.1145/2220365.2220367>
 - [12] Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. *Log. Methods Comput. Sci.* **8**(1) (2012), [https://doi.org/10.2168/LMCS-8\(1:24\)2012](https://doi.org/10.2168/LMCS-8(1:24)2012)
 - [13] Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Asynchronous sessions with input races. In: Carbone, M., Neykova, R. (eds.) Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022, EPTCS, vol. 356, pp. 12–23 (2022), <https://doi.org/10.4204/EPTCS.356.2>
 - [14] Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: Freund, S.N., Yahav, E. (eds.) PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021, pp. 237–251, ACM (2021), <https://doi.org/10.1145/3453483.3454041>
 - [15] Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. *Inf. Comput.* **202**(2), 166–190 (2005), <https://doi.org/10.1016/j.ic.2005.05.006>
 - [16] Charalambides, M., Dinges, P., Agha, G.A.: Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.* **115–116**, 100–126 (2016), <https://doi.org/10.1016/j.scico.2015.10.006>
 - [17] Chen, R., Balzer, S., Toninho, B.: Ferrite: A judgmental embedding of session types in rust. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6–10, 2022, Berlin, Germany, LIPIcs, vol. 222, pp. 22:1–22:28, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), <https://doi.org/10.4230/LIPIcs.ECOOP.2022.22>
 - [18] Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. *Math.*

- Struct. Comput. Sci. **26**(2), 238–302 (2016), <https://doi.org/10.1017/S0960129514000188>
- [19] Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. *Theor. Comput. Sci.* **802**, 38–66 (2020), <https://doi.org/10.1016/j.tcs.2019.07.005>
 - [20] Cruz-Filipe, L., Montesi, F., Peressotti, M.: Communications in choreographies, revisited. In: Haddad, H.M., Wainwright, R.L., Chbeir, R. (eds.) *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09–13, 2018*, pp. 1248–1255, ACM (2018), <https://doi.org/10.1145/3167132.3167267>
 - [21] Dagnino, F., Giannini, P., Dezani-Ciancaglini, M.: Deconfined global types for asynchronous sessions. In: Damiani, F., Dardha, O. (eds.) *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021*, *Proceedings, Lecture Notes in Computer Science*, vol. 12717, pp. 41–60, Springer (2021), https://doi.org/10.1007/978-3-030-78142-2_3
 - [22] Das, A., Balzer, S., Hoffmann, J., Pfenning, F., Santurkar, I.: Resource-aware session types for digital contracts. In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21–25, 2021*, pp. 1–16, IEEE (2021), <https://doi.org/10.1109/CSF51468.2021.00004>
 - [23] Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012*, *Proceedings, Lecture Notes in Computer Science*, vol. 7211, pp. 194–213, Springer (2012), https://doi.org/10.1007/978-3-642-28869-2_10
 - [24] Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. *Log. Methods Comput. Sci.* **8**(4) (2012), [https://doi.org/10.2168/LMCS-8\(4:6\)2012](https://doi.org/10.2168/LMCS-8(4:6)2012)
 - [25] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity OS. In: Berbers, Y., Zwaenepoel, W. (eds.) *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18–21, 2006*, pp. 177–190, ACM (2006), <https://doi.org/10.1145/1217935.1217953>
 - [26] Gazagnaire, T., Genest, B., Hélouët, L., Thiagarajan, P.S., Yang, S.: Causal message sequence charts. In: Caires, L., Vasconcelos, V.T. (eds.) *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3–8, 2007*, *Proceedings, Lecture Notes in Computer Science*, vol. 4703, pp. 166–180, Springer (2007), https://doi.org/10.1007/978-3-540-74407-8_12
 - [27] Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. *Fundam. Inform.* **80**(1–3), 147–167 (2007), URL <http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09>

- [28] Genest, B., Muscholl, A.: Message sequence charts: A survey. In: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France, pp. 2–4, IEEE Computer Society (2005), <https://doi.org/10.1109/ACSD.2005.25>
- [29] Genest, B., Muscholl, A., Peled, D.A.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets, Advances in Petri Nets* [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned], *Lecture Notes in Computer Science*, vol. 3098, pp. 537–558, Springer (2003), https://doi.org/10.1007/978-3-540-27755-2_15
- [30] Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.* **72**(4), 617–647 (2006), <https://doi.org/10.1016/j.jcss.2005.09.007>
- [31] Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-contract for flexible multiparty session protocols. In: Ali, K., Vitek, J. (eds.) *36th European Conference on Object-Oriented Programming, ECOOP 2022*, June 6-10, 2022, Berlin, Germany, *LIPIcs*, vol. 222, pp. 8:1–8:28, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), <https://doi.org/10.4230/LIPICS.ECOOP.2022.8>
- [32] Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty languages: The choreographic and multitier cases (pearl). In: Møller, A., Sridharan, M. (eds.) *35th European Conference on Object-Oriented Programming, ECOOP 2021*, July 11-17, 2021, Aarhus, Denmark (Virtual Conference), *LIPIcs*, vol. 194, pp. 22:1–22:27, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), <https://doi.org/10.4230/LIPICS.ECOOP.2021.22>
- [33] Girard, J.: Linear logic. *Theor. Comput. Sci.* **50**, 1–102 (1987), [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4)
- [34] Giusto, C.D., Laversa, L., Lozes, É.: On the k-synchronizability of systems. In: Goubault-Larrecq, J., König, B. (eds.) *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, *Proceedings, Lecture Notes in Computer Science*, vol. 12077, pp. 157–176, Springer (2020), https://doi.org/10.1007/978-3-030-45231-5_9
- [35] Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* **6**(POPL), 1–27 (2022), <https://doi.org/10.1145/3498684>
- [36] Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) *CONCUR ’93*, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, *Proceedings, Lecture Notes in Computer Science*, vol. 715, pp. 509–523, Springer (1993), https://doi.org/10.1007/3-540-57208-2_35

- [37] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pp. 273–284, ACM (2008), <https://doi.org/10.1145/1328438.1328472>
- [38] Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Lecture Notes in Computer Science*, vol. 10202, pp. 116–133, Springer (2017), https://doi.org/10.1007/978-3-662-54494-5_7
- [39] Igarashi, A., Thiemann, P., Tsuda, Y., Vasconcelos, V.T., Wadler, P.: Gradual session types. *J. Funct. Program.* **29**, e17 (2019), <https://doi.org/10.1017/S0956796819000169>
- [40] Jacobs, J., Balzer, S., Krebbers, R.: Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.* **6**(POPL), 1–33 (2022), <https://doi.org/10.1145/3498662>
- [41] Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.* **6**(ICFP), 466–495 (2022), <https://doi.org/10.1145/3547638>
- [42] Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for rust. In: Bahr, P., Erdweg, S. (eds.) *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pp. 13–22, ACM (2015), <https://doi.org/10.1145/2808098.2808100>
- [43] Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: A coalgebraic view on regular and context-free session types. *ACM Trans. Program. Lang. Syst.* **44**(3), 18:1–18:45 (2022), <https://doi.org/10.1145/3527633>
- [44] Kobayashi, N.: A new type system for deadlock-free processes. In: *CONCUR, Lecture Notes in Computer Science*, vol. 4137, pp. 233–247, Springer (2006), https://doi.org/10.1007/11817949_16
- [45] Kouzapas, D., Gutkovas, R., Voinea, A.L., Gay, S.J.: A session type system for asynchronous unreliable broadcast communication. *CoRR abs/1902.01353* (2019), URL <http://arxiv.org/abs/1902.01353>
- [46] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978), <https://doi.org/10.1145/359545.359563>
- [47] Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 1137–1148, ACM (2018), <https://doi.org/10.1145/3180155.3180157>
- [48] Li, E., Stutz, F., Wies, T., Zufferey, D.: Complete multiparty session type projection with automata. In: Enea, C., Lal, A. (eds.) *Computer Aided*

- Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III, Lecture Notes in Computer Science, vol. 13966, pp. 350–373, Springer (2023), https://doi.org/10.1007/978-3-031-37709-9_17
- [49] Lindley, S., Morris, J.G.: Embedding session types in haskell. In: Mainland, G. (ed.) Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016, pp. 133–145, ACM (2016), <https://doi.org/10.1145/2976002.2976018>
 - [50] Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.* **309**(1-3), 529–554 (2003), <https://doi.org/10.1016/j.tcs.2003.08.002>
 - [51] Majumdar, R., Mukund, M., Stutz, F., Zufferey, D.: Generalising projection in asynchronous multiparty session types. In: Haddad, S., Varacca, D. (eds.) 32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference, LIPIcs, vol. 203, pp. 35:1–35:24, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), <https://doi.org/10.4230/LIPIcs.CONCUR.2021.35>
 - [52] Majumdar, R., Pirron, M., Yoshida, N., Zufferey, D.: Motion session types for robotic interactions (brave new idea paper). In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom, LIPIcs, vol. 134, pp. 28:1–28:27, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), <https://doi.org/10.4230/LIPIcs.ECOOP.2019.28>
 - [53] Mauw, S., Reniers, M.A.: High-level message sequence charts. In: Cavalli, A.R., Sarma, A. (eds.) SDL ’97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings, pp. 291–306, Elsevier (1997)
 - [54] Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in f#. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria, pp. 128–138, ACM (2018), <https://doi.org/10.1145/3178372.3179495>
 - [55] Peng, W., Purushothaman, S.: Analysis of a class of communicating finite state machines. *Acta Informatica* **29**(6/7), 499–522 (1992), <https://doi.org/10.1007/BF01185558>
 - [56] Roychoudhury, A., Goel, A., Sengupta, B.: Symbolic message sequence charts. *ACM Trans. Softw. Eng. Methodol.* **21**(2), 12:1–12:44 (2012), <https://doi.org/10.1145/2089116.2089122>
 - [57] Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy, LIPIcs, vol. 56, pp. 21:1–21:28, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), <https://doi.org/10.4230/LIPIcs.ECOOP.2016.21>

- [58] Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Technical Report 6. Imperial College London (2018), URL <https://www.doc.ic.ac.uk/research/technicalreports/2018/DTRS18-6.pdf>
- [59] Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* **3**(POPL), 30:1–30:29 (2019), <https://doi.org/10.1145/3290343>
- [60] Stutz, F.: Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17–21, 2023, Seattle, Washington, United States, LIPIcs, vol. 263, pp. 32:1–32:31, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023), <https://doi.org/10.4230/LIPIcs.ECOOP.2023.32>
- [61] Stutz, F.: Implementability of Asynchronous Communication Protocols - The Power of Choice. Ph.D. thesis, Kaiserslautern University of Technology, Germany (2024), URL <https://kluedo.ub.rptu.de/frontdoor/index/index/docId/8077>
- [62] Stutz, F., D’Ousualdo, E.: An automata-theoretic basis for specification and type checking of multiparty protocols. *CoRR* **abs/2501.16977** (2025), <https://doi.org/10.48550/arXiv.2501.16977>
- [63] Stutz, F., Zufferey, D.: Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. In: Ganty, P., Monica, D.D. (eds.) Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21–23, 2022, EPTCS, vol. 370, pp. 194–212 (2022), <https://doi.org/10.4204/EPTCS.370.13>
- [64] Thiemann, P.: Intrinsically-typed mechanized semantics for session types. In: Komendantskaya, E. (ed.) Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7–9, 2019, pp. 19:1–19:15, ACM (2019), <https://doi.org/10.1145/3354166.3354184>
- [65] Thiemann, P., Vasconcelos, V.T.: Context-free session types. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016, pp. 462–475, ACM (2016), <https://doi.org/10.1145/2951913.2951926>
- [66] Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20–22, 2011, Odense, Denmark, pp. 161–172, ACM (2011), <https://doi.org/10.1145/2003476.2003499>
- [67] Toninho, B., Yoshida, N.: Depending on session-typed processes. In: Baier, C., Lago, U.D. (eds.) Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings, Lecture

- Notes in Computer Science, vol. 10803, pp. 128–145, Springer (2018), https://doi.org/10.1007/978-3-319-89366-2_7
- [68] Torre, S.L., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, Lecture Notes in Computer Science, vol. 4963, pp. 299–314, Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_21
 - [69] Union, I.T.: Z.120: Message sequence chart. Tech. rep., International Telecommunication Union (October 1996), URL <https://www.itu.int/rec/T-REC-Z.120>
 - [70] Viering, M., Hu, R., Eugster, P., Ziarek, L.: A multiparty session typing discipline for fault-tolerant event-driven distributed programming. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021), <https://doi.org/10.1145/3485501>
 - [71] Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30–31, 2013, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8358, pp. 22–41, Springer (2013), https://doi.org/10.1007/978-3-319-05119-2_3
 - [72] Zielonka, W.: Notes on finite asynchronous automata. RAIRO Theor. Informatics Appl. **21**(2), 99–135 (1987), <https://doi.org/10.1051/ITA/1987210200991>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Named Arguments as Intersections, Optional Arguments as Unions

Yaozhu Sun^(✉) and Bruno C. d. S. Oliveira

The University of Hong Kong, Hong Kong, China
{yzsun,bruno}@cs.hku.hk

Abstract. Named and optional arguments are prevalent features in many mainstream programming languages, enhancing code readability and flexibility. Despite widespread use, their formalization has not been extensively studied. This paper bridges this gap by presenting a type-safe foundation for named and optional arguments using intersection and union types. We begin by identifying a critical type-safety issue in popular static type checkers for Python and Ruby, particularly in handling first-class named arguments in the presence of subtyping. Our solution involves rewriting call sites to ensure type safety, which we formalize through a translation into a core calculus called λ_{iu} . The type safety of the translation is proven using the Coq proof assistant. The practical implementation of our approach in the CP language validates our theoretical contributions. Furthermore, we informally discuss how our approach could be adapted to encode named and optional arguments in other existing languages.

1 Introduction

The λ -calculus, introduced by Alonzo Church [5], shows how to model computation solely with function abstraction and application. For example, natural numbers, boolean values, pairs, and lists, as well as various operations on them, can be represented by higher-order functions via Church encoding. In the λ -calculus, a function only has one parameter and can only be applied to one argument. Many programming languages in the ML family inherit this feature. If more than one argument is desired in those languages, we need to create a sequence of functions, each with a single argument, and perform an iteration of applications. This idea is called *currying*. Currying brings brevity to functional programming and naturally allows partial application, but it usually limits the flexibility of function application. For example, we cannot pass arguments in a different order nor omit some of them by providing default values. Both demands are not rare in practical programming and can be met in a language that supports *named* and *optional* arguments. Named arguments also largely improve the readability of function calls. For example, it is unclear which is the source and which is the destination in `copy(x, y)`, while `copy(to: x, from: y)` is self-explanatory.

<pre>def exp(x, base=math.e): return base ** x exp(10, 2) <i>#= exp(x=10, base=2) = 1024</i> exp(base=2, x=10) <i>#= 1024</i> exp(x=10) <i>#= e¹⁰</i> args = { "base": 2, "x": 10 } exp(**args) <i>#= exp(base=2, x=10) = 1024</i></pre>	<pre>def exp(x:, base: Math::E) base ** x end exp(10, 2) <i># ArgumentError!</i> exp(base: 2, x: 10) <i>#= 1024</i> exp(x: 10) <i>#= e¹⁰</i> args = { base: 2, x: 10 } exp(**args) <i>#= 1024</i></pre>
(a) The Python way.	(b) The Ruby way.

Fig. 1: Named arguments in Python and Ruby.

Named arguments are widely supported in mainstream programming languages, such as Python, Ruby, OCaml, C#, and Scala, just to name a few. The earliest instance, to the best of our knowledge, is Smalltalk, where every method argument *must* be associated with a *keyword* (i.e. an external name). In other words, there are no positional arguments (i.e. arguments with no keywords) in Smalltalk. The syntax of modern languages is usually less rigid, so programmers can choose whether to attach keywords to arguments or not. There are two ways to reconcile positional and named arguments. One way, employed by Python and shown in Fig. 1a, is to make parameter names in a function definition as non-mandatory keywords. Thus, every argument can be passed with or without keywords by default. As shown in the Python code, `exp(10, 2)` is equivalent to `exp(x=10, base=2)`. To reconcile the two forms in the same call, a restriction is imposed that all named arguments must follow positional ones. The other way, shown in Fig. 1b and used in Ruby, is to strictly distinguish named arguments from positional ones. When defining a Ruby function, a keyword parameter should always end with a colon even if it does not have a default value. By this means, they are syntactically distinct from positional parameters, and their keywords cannot be omitted in a function call. There is also a restriction in Ruby that all named arguments must follow positional ones in both function definitions and call sites. The two kinds of arguments are usually used in different scenarios: positional arguments are used when the number of arguments is small and the order is clear, while named arguments are used in more complex cases especially when settings or configurations are involved.

More interestingly, named arguments are *first-class* values in Python and Ruby: they can be assigned to a variable. As shown at the bottom of Fig. 1, the variable `args` stores the two arguments named `base` and `x`, and we can later pass it to `exp` by unpacking it with `**` (sometimes called the splat operator). In fact, `args` is a dictionary in Python and similarly a hash in Ruby. Thus, first-class named arguments can be manipulated and passed around like standard data structures. This feature is widely used in Python and Ruby.

Table 1: Named arguments with different design choices in different languages.

	Smalltalk	Python	Ruby	Racket	OCaml	C#	Scala	Dart	Swift	This paper
Commutativity	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓
Optionality	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Currying	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Distinctness	<i>n.a.</i> [†]	✗	✓	✓	✓	✗	✗	✓	✓	✓
First-class value	✗	✓	✓	✗ [‡]	✗	✗	✗	✗	✗	✓
Static typing	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
Soundness proof	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓

[†] Smalltalk does not support positional arguments at all.

[‡] Racket’s support for first-class named arguments is limited and forbids commutativity.

Including the distinctness and first-class values illustrated above, we have identified five important design choices found in existing languages that support named arguments:

1. *Commutativity*: whether the order of (actual) arguments can be different from that of (formal) parameters originally declared.
2. *Optionality*: whether some arguments can be omitted in a function call if their default values are predefined.
3. *Currying*: whether a function that takes more than one argument is always converted into a chain of functions that each take a single argument.
4. *Distinctness*: whether named arguments are distinct from positional ones in how they are defined and passed.
5. *First-class value*: whether named arguments are first-class values.

As shown in Table 1, the first two properties hold for most mainstream programming languages, with Smalltalk and Swift being two exceptions. Commutativity and optionality are so useful that we believe they should not be compromised. Concerning the third point, OCaml is the only language that manages to reconcile currying with commutativity, though at the cost of introducing a very complicated core calculus. We agree that currying is very useful when we use normal positional arguments, but we argue here that currying can be temporarily dropped when we use named arguments because the most common use case for named arguments is to represent a whole chunk of parameters like settings or configurations. The fourth design, distinctness, is endorsed by Ruby, Racket, OCaml, Dart, and Swift. It improves the readability of call sites to enforce keywords whenever arguments are defined to be named. We advocate distinctness in this paper also because it simplifies the language design and allows us to focus on more important topics, especially type safety with first-class named arguments.

Although named arguments are ubiquitous, they have not attracted enough attention in the research of programming languages. Among the few related papers, the work by Garrigue et al. [1,9,12] formalizes a label-selective λ -calculus and eventually applies it to OCaml [11]. Another work by Rytz and Odersky [26]

discusses the design of named and optional arguments in Scala, but it mainly focuses on practical aspects. The core features of Scala are formalized in a family of DOT calculi, but named arguments are never included. The support for named arguments is implemented as macros in Racket [7]. So their extension is more like userland syntactic sugar and requires no changes to the core compiler. Haskell does not support named arguments natively, but the paradigm of *named arguments as records* is folklore. We will discuss OCaml, Scala, Racket, and Haskell in detail in Section 5. In short, named arguments are implemented in an ad-hoc manner and are not well founded from a type-theoretic perspective in most languages, especially object-oriented ones. Only OCaml and this paper provide *soundness proofs* for the feature of named arguments.

An important issue that has not been explored in the literature is the interaction between subtyping and first-class named arguments. A naive design can easily lead to a type-safety issue. We will show in Section 2.2 that the most widely used optional type checker for Python, mypy [13], fails to detect a type-unsafe use of first-class named arguments. The same issue also exists in Ruby with Steep [16] or Sorbet [28]. It arises from subtyping hiding some arguments from their static type and bypassing the type checking for optional arguments. As a result, an optional argument may have an unexpected type at run time, which leads to a runtime error.

In this paper, we present a type-safe foundation for named and optional arguments. At the heart of our approach is the translation into a core calculus called λ_{iu} , which features *intersection and union types* [2,6,8]. Our approach supports first-class named arguments like Python and Ruby, but the type-safety issue is addressed by us. The λ_{iu} calculus has been shown to be type-sound [23], and we show that our translation from our source language into λ_{iu} is type-safe. Thus, we establish the type safety of our approach. In addition, our design has recently been incorporated into the CP language [33].

In summary, the contributions of this paper are:

- We identify a type-safety issue with first-class named arguments in the presence of subtyping and propose a solution based on call site rewriting.
- We demonstrate how a minimal language with named and optional arguments can be translated to a core calculus with intersection and union types.
- We formalize the translation as an elaboration semantics and prove its type safety using the Coq proof assistant.
- We validate our theoretical contributions by implementing our approach in the CP language and showcasing a practical example.
- We conduct a survey of named arguments in existing programming languages and discuss the best practice for *named arguments as records* in Haskell.

2 Named and Optional Arguments: The Bad Parts

Since named and optional arguments are not well studied in most languages, the ad-hoc mechanisms employed in those languages may sometimes surprise programmers or even cause safety issues.

2.1 Gotcha! Mutable Default Arguments in Python

Let us consider a simple Python function that appends an element to a list. We provide a default value for the list, which is an empty list:

```
def append(x, xs=[]):
    xs.append(x)
    return xs
```

append(1) *#= [1]*
append(2) *#= [1, 2]*

After calling `append(1)`, we get the expected result `[1]`. However, continuing to call `append(2)` gives us `[1, 2]` instead of `[2]`. This is because Python only evaluates the default value once when the function is defined, so the same list initialized for `xs` is shared across different calls to `append`. When calling `append(2)`, the default value for `xs` is no longer an empty list but the list that has been modified by the previous call `append(1)`.

This issue, while seemingly minor, highlights the importance of understanding the semantics of default arguments. Our design strives to avoid such surprises, following the principle of least astonishment, yet this is not our main focus. We will discuss the more critical issue about type safety next.

2.2 Caution! Type Safety with First-Class Named Arguments

As we have shown in [Fig. 1](#), quite a few languages, especially dynamically typed ones like Python and Ruby, treat named arguments as first-class values. This feature is particularly helpful for passing settings because they are usually stored in a separate configuration file. We can read the settings from the file and pass them as named arguments using the `**` operator. For example, we can find such code in Python to run a web server:

```
class App: # from a web server library
    def run(self, host: str, port: int, debug: bool = False):
        assert isinstance(debug, bool) # actual code omitted...

args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug=True)
```

Although Python is dynamically typed, there is continuous effort in the Python community to improve the detection of type errors earlier in the development process, primarily through static analysis. There is an optional static type checker for Python called `mypy` [13]. In the example above, we make use of *type hints*, introduced in Python 3.5, to specify the types of the parameters and the return value of the `run` method. The type hints have no effect at run time but can be used by external tools like `mypy` to statically check if the code is well-typed. Perhaps surprisingly, the code above cannot pass `mypy`'s type checking, because the type inferred for `args` (i.e. `dict[str, object]`) is not precise enough. The type checker needs to know what keys `args` exactly has and what types the values associated with those keys have, in order to make sure that `**args` is compatible with the parameters of `app.run`.

Fortunately, `TypedDict` is added in Python 3.8 to represent a specific set of keys and their associated types. By default, every specified key is required, except when it is marked as `NotRequired`, which is a type qualifier added in Python 3.11. With `TypedDict` and `NotRequired`, we can now define a precise dictionary type for `args` that passes mypy's type checking:

```
class Args(TypedDict):
    host: str
    port: int
    debug: NotRequired[bool]

args0: Args = { "host": "0.0.0.0", "port": 80, "debug": True }
app.run(**args0) # type-checks in mypy
args1: Args = { "host": "0.0.0.0", "port": 80 }
app.run(**args1) # type-checks in mypy, too
```

The mypy type checker will raise an error if we provide an argument with an incompatible type, such as a string for the `debug` key:

```
class In(TypedDict):
    host: str
    port: int
    debug: str

args2: In = { "host": "0.0.0.0", "port": 80, "debug": "Oops!" }
app.run(**args2) # TypeError: Argument "debug"
# has incompatible type "str"; expected "bool" [arg-type]
```

However, mypy's type system is not completely type-safe. We can create a function `f` that takes a dictionary with three keys specified in type `In` and returns a dictionary with only two keys specified in type `Out`. The function type-checks in mypy because type `In` is compatible whenever type `Out` is expected. Roughly speaking, it means that `In` is a subtype of `Out`. Then we can use `f` to forget the `debug` key in the static type:

```
class Out(TypedDict):
    host: str
    port: int

def f(args: In) → Out: return args

args3 = f(args2) # still contains { "debug": "Oops!" }
app.run(**args3) # type-checks in mypy, but has a runtime error!
```

Here `args3` has type `Out` without the `debug` key specified. From a static view-point, `args3` only has two keys `host` and `port`, which are compatible with the parameters of `app.run` since `debug` is optional and has a default value. That is why `app.run(**args3)` type-checks in mypy. However, at run time, the `debug` key is still present in `args3`, so the string `"Oops!"` is passed as a named argument

to `app.run`, which originally expects a boolean value. This results in a runtime error since there is an assertion in `app.run` to ensure that `debug` is boolean.

This issue is not unique to Python and mypy. We have reproduced nearly the same issue in Ruby with two popular type checkers, namely Steep [16] and Sorbet [28], which is illustrated in Appendices A&B.

In conclusion, subtyping can lead to a fundamental type-safety issue when dealing with first-class named and optional arguments. In essence, the following subsumption chain is questionable:

```
{ host: str, port: int, debug: str }
<: { host: str, port: int }
<: { host: str, port: int, debug?: bool }
```

Following this chain bypasses mypy’s type compatibility checking for the `debug` key. Next we will show how to break the chain and address the type-safety issue.

3 Our Type-Safe Approach

In this section, we informally present how we translate named and optional arguments into a core language with intersection and union types, while retaining type safety. We start by introducing the core language constructs that we need. Then we illustrate our translation scheme by example and demonstrate how it recovers type safety. After that, we showcase a practical example in the CP language, which has incorporated our approach to support named and optional arguments. Finally, we discuss how our translation scheme can be applied to other languages.

3.1 Core Language

The core language features intersection and union types, which establish an elegant duality in the type system. A value of the intersection type $A \wedge B$ can be assigned both A and B , whereas a value of the union type $A \vee B$ can be assigned either A or B . Intersection and union types correspond to the logical conjunction and disjunction respectively. Similar calculi are widely studied [2,6,8] and provide a well-understood foundation for named and optional arguments.

Named Arguments as Intersections. Named arguments are translated to multi-field records. However, the core language does not support multi-field records directly. There are only single-field records in the core language, and multiple fields are represented as intersections of single-field record types. For example, $\{x : \mathbb{Z}\} \wedge \{y : \mathbb{Z}\}$ represents a record type with two integer fields x and y . With intersection types, width subtyping for record types comes for free, and permutations of record fields are naturally allowed [24].

At the term level, a merge operator [6,23] (denoted by $,$ in this paper) is used to concatenate multiple single-field records to form multi-field records, reminiscent of Forsythe [24]. For example, $\{x = 1\}, \{y = 2\}$ forms a two-field record from two single-field records.

Optional Arguments as Unions. Optional arguments are translated to nullable types. A nullable type is not implicit in the core language but is represented as a union with the null type [18]. For example, an optional integer argument named z is translated to $\{z : \mathbb{Z} \vee \mathbf{Null}\}$.

At the term level, a type-based switch expression [8,23] is used to scrutinize a term of a union type, reminiscent of ALGOL 68 [32]. For example, `switch  $z$  case  $\mathbb{Z} \Rightarrow e_1$  case  $\mathbf{Null} \Rightarrow e_2$` returns e_1 if z is an integer or e_2 if `null`.

3.2 Translation by Example

Let us review the previous Python function in Section 2.1 that appends an element to a list, which defaults to an empty list:

```
def append(x: int, xs: list[int] = []): ...
```

The function will be translated to a core function as follows:

```
append =  $\lambda args : \{x : \mathbb{Z}\} \wedge \{xs : [\mathbb{Z}] \vee \mathbf{Null}\}.$ 
      let  $x = args.x$  in
      let  $xs = \mathbf{switch} \ args.xs \ \mathbf{as} \ xs \ \mathbf{case} \ \mathbb{Z} \Rightarrow xs \ \mathbf{case} \ \mathbf{Null} \Rightarrow []$  in
      ...
```

Here we can see that the default value (i.e. the empty list) is not shared across different calls to `append` because the default value is evaluated within the function body. Therefore, calling `append(x=1)` will consistently return `[1]` instead of surprisingly modifying the default value. This design leads to less astonishment and more predictable behavior.

Since we translate named parameter types to record types, we correspondingly translate named arguments to records. For example, the function call `append(x=1, xs=[0])` will be translated to `append( $\{x = 1\}, \{xs = [0]\})$ .`

Rewriting Call Sites. More importantly, we also rewrite call sites to add null values for absent optional arguments. For example, the function call `append(x=1)` will be rewritten and translated to `append( $\{x = 1\}, \{y = \mathbf{null}\})$ .`

Dependent Default Values. Another advantage of our translation scheme is that it naturally allows default values to depend on earlier arguments. Python and Ruby do not support dependent default values, but this feature can be useful in some practical scenarios. For example, when setting up I/O, we may want to output error messages to the same stream as `out` by default:

```
def setIO(in_, out, err = out): ...
```

The variable `out` can be used in the default value of `err` because it has been brought into scope by the previous `let-in` binding:

```
let out = args.out in
let err = switch args.err as err case IO  $\Rightarrow$  err case Null  $\Rightarrow$  out in
```

3.3 Recovering Type Safety

The type safety of our translation scheme is essentially guaranteed by call site rewriting. Besides adding null values for absent optional arguments, we also sanitize arguments to ensure that they are expected from the parameter list. Since named arguments are first-class and can be passed as a variable, we may not have literals like `append(x=1, xs=[0])` but splats like `append(**args)`. So the matching between (formal) parameters and (actual) arguments is performed based on their static types:

- If `args` has type $\{x : \mathbb{Z}\} \wedge \{xs : [\mathbb{Z}]\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=args.xs)`.
- If `args` only has type $\{x : \mathbb{Z}\}$, the call site will be rewritten to something equivalent to `append(x=args.x, xs=null)`.

For the `append` function, no other cases can pass the sanitization process.

Let us review the previous type-unsafe Python example in [Section 2.2](#):

```
def f(args: In) → Out: return args
args = f({ "host": "0.0.0.0", "port": 80, "debug": "Oops!" })
app.run(**args) #= app.run(host="0.0.0.0", port=80, debug="Oops!")
```

Recall that `args` has type `Out`, which is similar to `{ host: str, port: int }`. The `debug` key is forgotten in the static type but is still present at run time. It passes mypy's type checking but raises a runtime error. In our translation scheme, the call site will be rewritten to the following form based on the type of `args` (i.e. `Out`):

```
app.run(host=args.host, port=args.port, debug=null)
```

Therefore, type safety is recovered in our translation scheme.

Takeaways. There are two important observations from our translation scheme:

1. `{ required: A, optional?: B }` is not equivalent to `{ required: A }` because the former contains more information that prevents `optional` from being associated with other types than `B`. In other words, the `optional` argument can be absent, but if it is present, it must have type `B`.
2. Corresponding to the above observation at the type level, we explicitly pass a null value as an optional argument if it is statically missing. The null value fills the position of a potentially forgotten argument that may have a wrong type. In other words, we implement the splat operator as per the static type of named arguments.

3.4 Implementation in the CP Language

Our approach to named and optional arguments has been implemented in the CP language, a statically typed language for *compositional programming* [33]. CP supports not only intersection and union types but also the merge operator

```

-- from a SVG library in CP
SVG: { width: Int; height: Int } → [Element] → Graphic;  -- <svg>
Rect: { x: Int; y: Int; width: Int; height: Int
      ; rx?: Int; ry?: Int; color?: String } → Element;  -- <rect>
.....
-- client code
fractal { level = 4; x: Int; y: Int; width: Int; height: Int } =
  let args = { level = level-1; width = width/3; height = height/3 } in
  let center = Rect (args,{ x = x + width/3; y = y + height/3
                          ; color = "white" }) in
  if level == 0 then [center]
  else fractal (args,{ x = x;                y = y })
    ++ fractal (args,{ x = x + width/3;    y = y })
    ++ fractal (args,{ x = x + width*2/3; y = y })
    ++ fractal (args,{ x = x;                y = y + height/3 })
    ++ [center]
    ++ fractal (args,{ x = x + width*2/3; y = y + height/3 })
    ++ fractal (args,{ x = x;                y = y + height*2/3 })
    ++ fractal (args,{ x = x + width/3;    y = y + height*2/3 })
    ++ fractal (args,{ x = x + width*2/3; y = y + height*2/3 });
init = { x = 0; y = 0; width = 600; height = 600; color = "black" };
main = SVG init ([Rect init] ++ fractal init);

```

Fig. 2: Sierpiński carpets implemented in the CP language.

and type-based switch expression. The implementation of named and optional arguments in CP is a direct application of our translation scheme.

More interestingly, the sanitization process during call site rewriting comes for free because CP employs a *coercive* semantics for subtyping [14]. For example, a subtyping relation between `{ host: str, port: int, debug: str }` and `{ host: str, port: int }` implies a coercion function from subtype to supertype. In CP, such coercions are implicitly inserted to remove the forgotten fields (e.g. `debug` in this case). Therefore, the only remaining work is to add null values for absent optional arguments.

To demonstrate the use of named and optional arguments in CP, we show a fractal example in Fig. 2, which is adapted from previous work on CP's application to embedded DSLs [29]. The code makes use of named and optional arguments a lot, including both the `SVG/Rect` constructors from the library and the `fractal` function defined by the client. For example, `fractal` has five named arguments (`level`, `x`, `y`, `width`, and `height`), among which `level` is optional with a default value of 4.

It is worth noting that named arguments are used as first-class values in the CP code. On the first line of the `fractal` body, we store three fields `level`, `width`, and `height` in a variable `args`, which are shared arguments for later calls. When constructing the center rectangle, we merge `args` with three more fields `x`, `y`, and `color` to form a full set of named arguments we need for the

Rect constructor. When recursively calling **fractal**, we pass **args** merged with different **x** and **y** values to draw the eight sub-copies. In the **main** function, we also use a variable **init** to avoid repeating the same set of arguments for **SVG**, **Rect**, and **fractal** calls. The ****** operator is not needed in CP when passing first-class named arguments. Note that the parameter lists of these three constructors/functions are not completely the same, but we can still use a larger set of named arguments to cover all the cases. This is possible because CP allows subtyping for named arguments while retaining type safety.

3.5 Applications to Other Languages

Although we base our translation scheme on a core language with intersection and union types for type-theoretic solidness and elegance, it can work for a wider range of languages. We discuss the alternatives to intersections and unions below.

Alternative to Intersections. Record types have existed long before intersection types were invented. In practice, multi-field records are rarely represented as intersections of single-field records. For example, *Software Foundations* [22] demonstrates how to directly model multi-field records and define depth, width, and permutation subtyping without intersections, though their formalization is more complex than ours.

There is a merge operator in our translation scheme, but we only use it to construct multi-field records statically. Although the merge operator can be powerful if we want to construct first-class named arguments at run time like in CP, its absence does not disable our translation scheme. In other words, we only assume a simplified version that does not merge terms dynamically.

Alternative to Unions. Nullable types are rarely represented as unions with the null type too. For example, C#, Kotlin, and Dart support nullable types as a primitive data structure. Putting a question mark behind any type makes it nullable in these languages (e.g. **int?**).

No matter how a nullable type is represented, there is usually some expression that can check whether a nullable value is null or not. For example, C# provides the **is** operator to examine the runtime type, which is generally known as type introspection and is similar to the type-based switch. C# also provides the null coalescing operator **??** and simplifies the common pattern **switch e as x case A ⇒ x case Null ⇒ d as e??d** for nullable values.

Dynamically Typed Languages. It may be surprising at first sight that dynamically typed languages can benefit from our work with static typing, but recall that the type-safety issue in Section 2.2 was found in Python. Nowadays, popular dynamically typed languages have been retrofitted with gradual typing. For example, Python has type hints and mypy [13], Ruby has RBS and Steep [16], JavaScript gets typed by TypeScript [17], and Lua gets typed by Luau [25]. All of

these typed versions support record-like and union types, and all except Python also support intersection types. Our translation scheme can almost directly apply to these languages. For a concrete example, we show how the aforementioned `exp` function can be encoded in TypeScript:

```
function exp(args: { x: number } & { base: number|null }) {
  let x = args.x;
  let base = (typeof args.base === "number") ? args.base : Math.E;
  return Math.pow(base, x);
}
exp({ base: 2, x: 10 })      // = 1024
exp({ x: 10, base: null })  // = e^10
```

The code is almost the same as in [Section 3.2](#). Note that the `typeof` operator is the standard way to perform type introspection in TypeScript, and the type of `args.base` is refined from the `number|null` to `number` in the true-branch. We assume the call sites have been rewritten in the code above. In this manner, named and optional arguments can be added to TypeScript as syntactic sugar.

Although we have discussed several alternatives to intersection and union types, we believe that if a language is designed from scratch, our approach is a good choice. Intersection and union types not only subsume multi-field record and nullable types but also provide a solid and elegant foundation for other advanced features, such as function overloading and heterogeneous data structures. Castagna’s essay [3] is an excellent further reading on the beauty of programming with intersection and union types.

4 Formalization

In this section, we formalize the translation of named and optional arguments as an elaboration semantics. The target of elaboration is called λ_{iu} , and the source is called UAENA. We prove that the source language with named and optional arguments is type-safe via (1) the type soundness of the target calculus and (2) the type soundness of elaboration. All the theorems are mechanically proven using the Coq proof assistant.

4.1 The Target Calculus: λ_{iu}

λ_{iu} is an extension to the calculus in Chapter 5 of Rehman’s dissertation [23] with `null`, single-field records, and let-in bindings. The addition of let-in bindings is not essential because they can be desugared into lambda abstractions and applications:

$$\text{let } x = e_1 \text{ in } e_2 \quad \equiv \quad (\lambda x. e_2) e_1$$

However, we still have let-in bindings for the sake of readability, and this form of let-in bindings simplifies the rules of parameter elaboration (introduced later in [Fig. 5](#)). Another difference is that the original calculus uses the locally nameless representation [4] while ours directly uses names for bound variables.

Our changes to Rehman’s calculus are relatively trivial, and we do not touch the rules for intersection and union types. We will not discuss his design choices in this paper, because our focus is on the type soundness with the addition of **Null** and record types. We have proven in Coq that these extensions preserve type soundness.

Syntax of λ_{iu}

Types $A, B ::= \top \mid \perp \mid \mathbf{Null} \mid \mathbb{Z} \mid A \rightarrow B \mid \{\ell : A\} \mid A \wedge B \mid A \vee B$
 Expressions $e ::= \{\} \mid \mathbf{null} \mid n \mid x \mid \lambda x : A. e : B \mid e_1 e_2 \mid \{\ell : A = e\} \mid e.\ell$
 $\mid e_1, e_2 \mid \mathbf{switch} e_0 \mathbf{as} x \mathbf{case} A \Rightarrow e_1 \mathbf{case} B \Rightarrow e_2 \mid \mathbf{letin} e$

The types include the top type $\top$, the bottom type $\perp$, the null type **Null**, the integer type $\mathbb{Z}$, function types $A \rightarrow B$, record types $\{\ell : A\}$, intersection types $A \wedge B$, and union types $A \vee B$. **Null** is a unit type that has only one value **null**.

The expressions include the empty record $\{\}$, the null value **null**, integer literals n , variables x , lambda abstractions $\lambda x : A. e : B$, function applications $e_1 e_2$, record literals $\{\ell : A = e\}$, record projections $e.\ell$, merges e_1, e_2 , type-based switch expressions **switch** e_0 **as** x **case** $A \Rightarrow e_1$ **case** $B \Rightarrow e_2$, and let-in bindings *letin* e . The syntax of *letin* is as follows:

$$\mathbf{letin} ::= \mathbf{let} x = e \mathbf{in} \mid \mathbf{letin}_1 \circ \mathbf{letin}_2 \mid \mathbf{id}$$

The composition of two let-in bindings is denoted by $\mathbf{letin}_1 \circ \mathbf{letin}_2$, and an empty binding is denoted by **id**.

Subtyping. Fig. 3 shows the subtyping rules of λ_{iu} . The rules are standard for a type system with intersection and union types. Rule **SUB-TOP** shows that the

$A <: B$				(Subtyping)
SUB-NULL $\frac{}{\mathbf{Null} <: \mathbf{Null}}$	SUB-INT $\frac{}{\mathbb{Z} <: \mathbb{Z}}$	SUB-ARROW $\frac{A_2 <: A_1 \quad B_1 <: B_2}{A_1 \rightarrow B_1 <: A_2 \rightarrow B_2}$	SUB-RCD $\frac{A <: B}{\{\ell : A\} <: \{\ell : B\}}$	
SUB-AND $\frac{A <: B \quad A <: C}{A <: B \wedge C}$	SUB-ANDL $\frac{A <: C}{A \wedge B <: C}$	SUB-ANDR $\frac{B <: C}{A \wedge B <: C}$	SUB-OR $\frac{A <: C \quad B <: C}{A \vee B <: C}$	
SUB-ORL $\frac{A <: B}{A <: B \vee C}$	SUB-ORR $\frac{A <: C}{A <: B \vee C}$	SUB-TOP $\frac{}{A <: \top}$	SUB-BOT $\frac{}{\perp <: A}$	

Fig. 3: Subtyping of λ_{iu} .

top type $\top$ is a supertype of any type, and rule **SUB-BOT** shows that the bottom type $\perp$ is a subtype of any type. Rules **SUB-AND**, **SUB-ANDL**, and **SUB-ANDR** handle the subtyping for intersection types, while rules **SUB-OR**, **SUB-ORL**, and **SUB-ORR** are for union types. Rules **SUB-NULL** and **SUB-RCD** added by us are straightforward. We prove that the subtyping relation is reflexive and transitive.

Theorem 1 (Subtyping Reflexivity). $\forall A, A <: A$.

Theorem 2 (Subtyping Transitivity). *If $A <: B$ and $B <: C$, then $A <: C$.*

Typing. Fig. 4 shows the typing rules of λ_{iu} . The empty record $\{\}$ has the top type $\top$, as shown in rule **TYP-TOP**. Rule **TYP-MERGE** is the introduction rule for intersection types. Merging two functions is used for function overloading, and merging two records is used for record concatenation. Rule **TYP-SWITCH** is the elimination rule for union types. The type-based switch expression scrutinizes an expression having a union of the two scrutinizing types (i.e. $e_0 : A \vee B$). This premise ensures the exhaustiveness of the cases in the switch. The **as**-variable x is refined to type A in e_1 and to type B in e_2 . Rules **TYP-NULL**,

Typing contexts		$\Gamma ::= \cdot \mid \Gamma, x : A$	
$\Gamma \vdash e : A$		<i>(Typing)</i>	
TYP-TOP	TYP-NULL	TYP-INT	TYP-VAR
$\frac{}{\Gamma \vdash \{\} : \top}$	$\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Null}}$	$\frac{}{\Gamma \vdash n : \mathbb{Z}}$	$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$
TYP-ABS	TYP-APP	TYP-RCD	
$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A. e : B) : A \rightarrow B}$	$\frac{\Gamma \vdash e_1 : A \rightarrow B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1 e_2 : B}$	$\frac{\Gamma \vdash e : A}{\Gamma \vdash \{\ell : A = e\} : \{\ell : A\}}$	
TYP-PRJ	TYP-LET	TYP-MERGE	
$\frac{\Gamma \vdash e : \{\ell : A\}}{\Gamma \vdash e.\ell : A}$	$\frac{\Gamma \vdash \mathit{letin} \dashv \Gamma' \quad \Gamma' \vdash e : A}{\Gamma \vdash \mathit{letin} e : A}$	$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1, e_2 : A \wedge B}$	
TYP-SWITCH		TYP-SUB	
$\frac{\Gamma \vdash e_0 : A \vee B \quad \Gamma, x : A \vdash e_1 : C \quad \Gamma, x : B \vdash e_2 : C}{\Gamma \vdash \mathbf{switch} e_0 \mathbf{as} x \mathbf{case} A \Rightarrow e_1 \mathbf{case} B \Rightarrow e_2 : C}$		$\frac{\Gamma \vdash e : A \quad A <: B}{\Gamma \vdash e : B}$	

Fig. 4: Typing of λ_{iu} .

TYP-RCD, and **TYP-PRJ** added by us are straightforward. Rule **TYP-LET** uses an auxiliary judgment $\Gamma \vdash \text{letin} \dashv \Gamma'$ (defined in Appendix C) to obtain the typing context for the body of the let-in binding. For example, if e_1 has type A , then **let** $x = e_1$ **in** e_2 adds $x : A$ to the typing context before type-checking e_2 .

Dynamic Semantics. We have a small-step operational semantics for λ_{iu} . The judgment $e \longrightarrow e'$ means that e reduces to e' in one step, and $e \longrightarrow^* e'$ is for multi-step reduction. We extend the original dynamic semantics by adding rules for records and projections. Similarly to the applicative dispatch for function applications in the original calculus, we add a relation called projective dispatch for record projections. For example, $(\{x = 1\}, \{y = 2\}).x$ reduces to $\{x = 1\}.x$ via projective dispatch to select the needed field.

Since the dynamic semantics of λ_{iu} is independent of the elaboration from UAENA to λ_{iu} , we omit the rules here but leave them in Appendix D. Note that the operational semantics is not commonplace in that it is type-directed and non-deterministic. Please refer to Rehman’s dissertation [23] for detailed explanations.

Theorem 3 (Progress). *If $\cdot \vdash e : A$, then either e is a value or $\exists e', e \longrightarrow e'$.*

Theorem 4 (Preservation). *If $\cdot \vdash e : A$ and $e \longrightarrow e'$, then $\cdot \vdash e' : A$.*

Putting progress and preservation together, we conclude that λ_{iu} is type-sound: a well-typed term can never reach a stuck state.

Corollary 1 (Type Soundness). *If $\cdot \vdash e : A$ and $e \longrightarrow^* e'$, then either e' is a value or $\exists e'', e' \longrightarrow e''$.*

4.2 The Source Calculus: UAENA

UAENA (*Unnamed Arguments Extended with Named Arguments*) is a minimal calculus with named and optional arguments. Although the calculus is small, named arguments are supported as first-class values and can be passed to or returned by a function. Besides functions with named arguments, UAENA also supports normal functions with positional arguments. The two kinds of functions are distinguished in the syntax, as seen in Ruby, Racket, OCaml, etc.

Syntax of UAENA

Types	$\mathcal{A}, \mathcal{B} ::= \mathbb{Z} \mid (\mathcal{A}) \rightarrow \mathcal{B} \mid \{\mathcal{P}\} \rightarrow \mathcal{B} \mid \{\mathcal{K}\}$
Named parameter types	$\mathcal{P} ::= \cdot \mid \mathcal{P}; \ell : \mathcal{A} \mid \mathcal{P}; \ell? : \mathcal{A}$
Named argument types	$\mathcal{K} ::= \cdot \mid \mathcal{K}; \ell : \mathcal{A}$
Expressions	$\epsilon ::= n \mid x \mid \lambda(x : \mathcal{A}). \epsilon \mid \lambda\{\rho\}. \epsilon \mid \epsilon_1 \epsilon_2 \mid \{\kappa\}$
Named parameters	$\rho ::= \cdot \mid \rho; \ell : \mathcal{A} \mid \rho; \ell = \epsilon$
Named arguments	$\kappa ::= \cdot \mid \kappa; \ell = \epsilon$

The types include the integer type $\mathbb{Z}$, normal function types $(\mathcal{A}) \rightarrow \mathcal{B}$, function types with named parameters $\{\mathcal{P}\} \rightarrow \mathcal{B}$, and (first-class) named argument types $\{\mathcal{K}\}$. The expressions include integer literals n , variables x , normal lambda abstractions $\lambda(x : \mathcal{A}). \epsilon$, lambda abstractions with named parameters $\lambda\{\rho\}. \epsilon$, function applications $\epsilon_1 \epsilon_2$, and (first-class) named arguments $\{\kappa\}$.

A named parameter type $\mathcal{P}$ can be required ($\ell : \mathcal{A}$) or optional ($\ell? : \mathcal{A}$). If a named parameter is optional, its default value must be provided in the function definition. For example, $\lambda\{x : \mathbb{Z}; y = 0\}. x + y$ has type $\{x : \mathbb{Z}; y? : \mathbb{Z}\} \rightarrow \mathbb{Z}$. A function with named parameters can only be applied to named arguments, which are basically a list of key-value pairs. For example, the previous function can be applied to $\{x = 1; y = 2\}$ or $\{x = 1\}$ or a variable having a compatible type. The variable case demonstrates the first-class nature of named arguments in UAENA.

Careful readers may notice that a named argument type can also serve as the parameter of a normal function. This also demonstrates the first-class nature of named arguments. But note that a normal function that takes named arguments is different from a function with named parameters. Consider the following two functions, the former of which is a function with named parameters and the latter is a normal function:

$$\begin{aligned} (\lambda\{x : \mathbb{Z}; y = 0\}. x + y) & : \{x : \mathbb{Z}; y? : \mathbb{Z}\} \rightarrow \mathbb{Z} \\ (\lambda(args : \{x : \mathbb{Z}; y : \mathbb{Z}\}). args) & : (\{x : \mathbb{Z}; y : \mathbb{Z}\}) \rightarrow \{x : \mathbb{Z}; y : \mathbb{Z}\} \end{aligned}$$

Although both functions can be applied to $\{x = 1; y = 2\}$, there are two main differences between them. First, optional parameters cannot be defined in a normal function. So we cannot provide $y = 0$ as a default value in the second function. Second, x and y are not brought into the scope of the function body in a normal function. So the only accessible variable is $args$ in the second function.

Elaboration. The type-directed elaboration from UAENA to λ_{iu} is defined at the top of Fig. 5. $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$ means that the source expression ϵ has type $\mathcal{A}$ and elaborates to the target expression e under the typing context Δ . Rules [ELA-ABS](#) and [ELA-APP](#) for normal functions are straightforward. In rule [ELA-NABS](#) for functions with named parameters, besides inferring the type of the function body ϵ and elaborating it to e , we generate let-bindings for the named parameters, which is delegated to the auxiliary judgment $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta'$. In rule [ELA-NAPP](#), there is also an auxiliary judgment $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ that rewrites call sites according to the parameter and argument types. Rules [ELA-NEMPTY](#) and [ELA-NFIELD](#) are used to elaborate named arguments.

Named Parameter Elaboration. As shown at the bottom of Fig. 5, $\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta'$ means that the named parameter ρ is inferred to have type $\mathcal{P}$ and elaborates to a series of let-in bindings letin , given that the named parameters correspond to the target bound variable x . In the meanwhile, the typing context Δ is extended with the types of the named parameters to form Δ' . Δ' is

Typing contexts		$\Delta ::= \cdot \mid \Delta, x : \mathcal{A}$
$\boxed{\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}$		(Elaboration)
ELA-INT	ELA-VAR	
$\frac{}{\Delta \vdash n : \mathbb{Z} \rightsquigarrow n}$	$\frac{x : \mathcal{A} \in \Delta}{\Delta \vdash x : \mathcal{A} \rightsquigarrow x}$	
ELA-ABS	ELA-APP	
$\frac{\Delta, x : \mathcal{A} \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda(x : \mathcal{A}). \epsilon : (\mathcal{A}) \rightarrow \mathcal{B} \rightsquigarrow \lambda x : \mathcal{A} . e : \mathcal{B} }$	$\frac{\Delta \vdash \epsilon_1 : (\mathcal{A}) \rightarrow \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \mathcal{A} \rightsquigarrow e_2}{\Delta \vdash \epsilon_1 \epsilon_2 : \mathcal{B} \rightsquigarrow e_1 e_2}$	
ELA-NABS	ELA-NAPP	
$\frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{B} \rightsquigarrow e}{\Delta \vdash \lambda\{\rho\}. \epsilon : \{\mathcal{P}\} \rightarrow \mathcal{B} \rightsquigarrow \lambda x : \mathcal{P} . \text{letin } e : \mathcal{B} }$	$\frac{\Delta \vdash \epsilon_1 : \{\mathcal{P}\} \rightarrow \mathcal{B} \rightsquigarrow e_1 \quad \Delta \vdash \epsilon_2 : \{\mathcal{K}\} \rightsquigarrow e_2 \quad \Delta \vdash_{e_2} \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'_2}{\Delta \vdash \epsilon_1 \epsilon_2 : \mathcal{B} \rightsquigarrow e_1 e'_2}$	
ELA-NEMPTY	ELA-NFIELD	
$\frac{}{\Delta \vdash \{\cdot\} : \{\cdot\} \rightsquigarrow \{\cdot\}}$	$\frac{\Delta \vdash \{\kappa\} : \{\mathcal{K}\} \rightsquigarrow e' \quad \Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\Delta \vdash \{\kappa; \ell = \epsilon\} : \{\mathcal{K}; \ell : \mathcal{A}\} \rightsquigarrow e', \{\ell : \mathcal{A} = e\}}$	
$\boxed{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta'}$		(Named parameter elaboration)
PELA-EMPTY		
$\frac{}{\Delta \vdash_x \cdot : \cdot \rightsquigarrow \mathbf{id} \dashv \Delta}$		
PELA-REQUIRED		
$\frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta'}{\Delta \vdash_x (\rho; \ell : \mathcal{A}) : (\mathcal{P}; \ell : \mathcal{A}) \rightsquigarrow \text{letin} \circ \mathbf{let} \ell = x.\ell \mathbf{in} \dashv \Delta', \ell : \mathcal{A}}$		
PELA-OPTIONAL		
$\frac{\Delta \vdash_x \rho : \mathcal{P} \rightsquigarrow \text{letin} \dashv \Delta' \quad \Delta' \vdash \epsilon : \mathcal{A} \rightsquigarrow e}{\Delta \vdash_x (\rho; \ell = \epsilon) : (\mathcal{P}; \ell? : \mathcal{A}) \rightsquigarrow \text{letin} \circ \mathbf{let} \ell = \mathbf{switch} x.\ell \mathbf{as} y \mathbf{case} \mathcal{A} \Rightarrow y \mathbf{case} \mathbf{Null} \Rightarrow e \mathbf{in} \dashv \Delta', \ell : \mathcal{A}}$		

 Fig. 5: Type-directed elaboration from UAENA to λ_{iu} .

$$\boxed{\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'} \quad (\text{Call site rewriting})$$

$$\begin{array}{c}
\text{PMAT-REQUIRED} \\
\frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| = e.\ell\}} \\
\\
\text{PMAT-PRESENT} \\
\frac{\mathcal{K} :: \ell \Rightarrow \mathcal{A} \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| \vee \mathbf{Null} = e.\ell\}} \\
\\
\text{PMAT-ABSENT} \\
\frac{\mathcal{K} :: \ell \nRightarrow \quad \Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'}{\Delta \vdash_e (\mathcal{P}; \ell? : \mathcal{A}) \diamond \mathcal{K} \rightsquigarrow e', \{\ell : |\mathcal{A}| \vee \mathbf{Null} = \mathbf{null}\}}
\end{array}$$

Fig. 6: Type-directed call site rewriting in UAENA.

used for typing the body of the function with named parameters. Rule **PELA-REQUIRED** simply generates **let** $\ell = x.\ell$ **in**, while rule **PELA-OPTIONAL** generates **let** $\ell = \mathbf{switch} \ x.\ell \mathbf{as} \ y \mathbf{case} \ |\mathcal{A}| \Rightarrow y \mathbf{case} \ \mathbf{Null} \Rightarrow e$ **in** to provide a default value e for the **Null** case.

Call Site Rewriting. As shown in Fig. 6, $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ means that if the parameter type $\mathcal{P}$ is compatible with the argument type $\mathcal{K}$, the target expression e , which corresponds to the named arguments, will be rewritten to e' . The compatibility check is based on the parameter type $\mathcal{P}$. Rule **PMAT-REQUIRED** handles the case where the argument is required, while rules **PMAT-PRESENT** and **PMAT-ABSENT** handle the cases where the optional argument with a specific type is present and where the optional argument is absent, respectively. The remaining case, where the optional argument is present but associated with a wrong type, is prohibited and cannot elaborate to any term. We have two more auxiliary judgments $\mathcal{K} :: \ell \Rightarrow \mathcal{A}$ and $\mathcal{K} :: \ell \nRightarrow$ to indicate that the argument type $\mathcal{K}$ contains a field ℓ with type $\mathcal{A}$ or $\mathcal{K}$ does not contain ℓ , whose definitions can be found in Appendix E.

Type Translation. As we have informally mentioned in Section 3.1, we translate named parameters to intersection types and optional parameters to union types. The rules for $|\cdot|$ can be found in Appendix F. Having defined the translation, we can prove the soundness of call site rewriting and elaboration.

Theorem 5 (Soundness of Call Site Rewriting). *If $\Delta \vdash_e \mathcal{P} \diamond \mathcal{K} \rightsquigarrow e'$ and $|\Delta| \vdash e : |\mathcal{K}|$, then $|\Delta| \vdash e' : |\mathcal{P}|$.*

Theorem 6 (Soundness of Elaboration). *If $\Delta \vdash \epsilon : \mathcal{A} \rightsquigarrow e$, then $|\Delta| \vdash e : |\mathcal{A}|$.*

With the two theorems above and the type soundness of λ_{iu} , we can conclude that UAENA is type-safe.

5 Discussion and Related Work

In this section, we first discuss OCaml, the only language we know of that has well-studied support for named and optional arguments, though its mechanism does not go well with higher-order functions. Then we briefly show how named arguments are handled very differently in Scala and Racket. After that, we illustrate how named arguments can be encoded as records in Haskell while not natively supported. Finally, we discuss two more approaches we find in record calculi [19,20]. We will also explain why all these approaches have drawbacks.

5.1 OCaml

OCaml did not support named arguments originally. Nevertheless, Garrigue et al. [1,9,12] conducted research on the label-selective λ -calculus and implemented it in OLabl [10], which extends OCaml with labeled and optional arguments, among others. All features of OLabl were merged into OCaml 3, despite subtle differences [11].

Here is an example of the exponential function defined in a labeled style:

```
let exp ?(base = Float.exp 1.0) x = base ** x
(* val exp : ?base:float → float → float *)
exp 10.0                (*= e10. *)
exp 10.0 ~base:2.0      (*= 1024. *)
(exp 10.0) ~base:2.0    (* TypeError! *)
```

In the definition of `exp`, `base` is an optional labeled parameter while `x` is a positional parameter. Changing `x` into a second labeled parameter will trigger an `unerasable-optional-argument` warning because OCaml expects that there should be a positional parameter after all optional parameters. This expectation is at the heart of how OCaml resolves the ambiguity introduced by currying.

For example, consider the function application `exp 10.0`. Is it a partially applied function or a fully applied one using the default value of `base`? Both interpretations are possible, but OCaml considers it to be a full application because the trailing positional argument `x` is given. The presence of the positional argument is used to indicate that the optional arguments before it can be replaced by their default values. However, this design may confuse users since `(exp 10.0) ~base:2.0` will raise a type error but `exp 10.0 ~base:2.0` will not. Partial application does not lead to an equivalent program in such situations.

Option Types. In OCaml, an optional argument is internally represented as an `option` type, which comprises two constructors: `None` and `Some`. Here is an equivalent definition for `exp`:

```
let exp ?(base : float option) x =
  let base = match base with
    | None → Float.exp 1.0
    | Some b → b in
  base ** x
(* val exp : ?base:float → float → float *)
exp 10.0 (*> exp 10.0 ~base:None *)
exp 10.0 ~base:2.0 (*> exp 10.0 ~base:(Some 2.0) *)
```

This encoding is similar to union types, but it depends on the `option` type in the standard library. Unfortunately, this specific kind of `option` is not a built-in type in many mainstream languages, especially in those languages that do not support algebraic data types.

Higher-Order Functions. A surprising gotcha in OCaml is that the commutativity breaks down when we pass a function with labeled arguments to another function. *Real World OCaml* [15] gives the following example:

```
let apply1 f (fst,snd) = f ~fst ~snd
(* val apply1 : (fst:'a → snd:'b → 'c) → 'a * 'b → 'c *)
let apply2 f (fst,snd) = f ~snd ~fst
(* val apply2 : (snd:'a → fst:'b → 'c) → 'b * 'a → 'c *)
let divide ~fst ~snd = fst / snd
(* val divide : fst:int → snd:int → int *)
apply1 divide (48,3) (*= 16 *)
apply2 divide (48,3)
(* TypeError: "divide" has type fst:int → snd:int → int
   but was expected of type snd:'a → fst:'b → 'c *)
```

Normally, the order of named arguments does not matter in OCaml, so it type-checks whether we call `divide ~fst ~snd` or `divide ~snd ~fst`. However, order matters when we pass `divide` to a higher-order function. That is why `apply1 divide` type-checks while `apply2 divide` does not. It turns out that the OCaml way of handling labeled arguments does not go well with other features like higher-order functions. Our approach scales better in this regard and the commutativity still holds in higher-order contexts via intersection subtyping.

In short, OCaml has a very powerful label-selective core calculus that reconciles commutativity and currying, but it is quite complicated and may hinder its integration with other language features. Another thing worth mentioning is that labeled arguments in OCaml are not first-class values, so they cannot be assigned to a variable or passed around by functions. In contrast to OCaml, our approach supports first-class named arguments and targets a minimal core calculus with intersection and union types, which is compatible with many popular languages like Python, Ruby, JavaScript, etc.

5.2 Scala

Rytz and Odersky [26] described the design of named and default arguments in Scala. Like in Python, parameter names in a method definition are non-mandatory keywords in Scala, and thus every argument can be passed with or without keywords. Furthermore, the parameter names are not part of the public interface of a method. This design is partly due to the backward compatibility with earlier versions of Scala, so the addition of named arguments will not break any existing code. As a result of the conservative treatment, named arguments are not first-class values in Scala and cannot be defined in an anonymous function. In short, named arguments are more like syntactic sugar in Scala and do not interact with the type system.

Below we show an example in Scala. In order to let the default value of `c` depend on `a` and `b`, we make the function partly curried:

```
def f(a: Int, b: Int)(c: Int = a+b) = c
f(b = 1+1, a = 1)()
```

The code will be translated to equivalent code without keywords or defaults:

```
def f(a: Int, b: Int)(c: Int) = c
def f$default$3(a: Int, b: Int): Int = a+b
{
  val x$1 = 1+1
  val x$2 = 1
  val x$3 = f$default$3(x$2, x$1)
  f(x$2, x$1)(x$3)
}
```

There are two things to note here. First, a new function `f$default$3` is generated for the default value of `c`, taking two parameters `a` and `b`. Second, the call site is translated to a series of variable assignments for each argument and a keyword-free call to `f` with arguments reordered. The whole call site is wrapped in a block to avoid polluting the namespace.

In conclusion, named arguments in Scala are handled in a very different way from OCaml and our approach. The Scala way is more syntactic than type-theoretic, so it is hard to do an apples-to-apples comparison with our approach.

5.3 Racket

Flatt and Barzilay [7] introduced keyword and optional arguments into Racket, which was known as PLT Scheme at that time. A keyword is prefixed with `#:` in syntax and is implemented as a new built-in type in Racket. Keyword arguments are supported by replacing `define`, `lambda`, and the core application form with newly defined macros that recognize keyword-argument forms. Here is an example of a function `f` with three keyword arguments `a`, `b`, and `c`, among which `c` is optional and defaults to `a+b`:

```
(define (f #:a a #:b b #:c [c (+ a b)]) c)
(f #:b (+ 1 1) #:a 1)
```

The function call with keywords seems hard to implement because it just lists the function and arguments in juxtaposition. In fact, an application form in Racket implicitly calls `#%app` in its lexical scope, so the support for keyword arguments is done by supplying an `#%app` macro. A new **keyword-apply** function is also defined to accept keyword arguments as first-class values. For example, we can rewrite the function call above as follows¹:

```
(keyword-apply f '(:a #:b) '(1 ,(+ 1 1)) '());OK!
(keyword-apply f '(:b #:a) '(, (+ 1 1) 1) '());Contract violation!
```

Note that we need to separate keywords and corresponding arguments into two lists. The third list is for positional arguments, so it is empty in this case. We cannot list keywords in arbitrary order: a contract violation will be signaled unless the keywords are sorted in alphabetical order. In other words, commutativity is lost for first-class keyword arguments in Racket.

In Typed Racket [31], Racket’s gradually typed sister language, `f` can be typed as $(\rightarrow^* (\# :a \text{ Number } \# :b \text{ Number}) (\# :c \text{ Number}) \text{ Number})$. The first list contains the required arguments (`#:a` and `#:b`), and the second list contains the optional ones (`#:c`). However, Typed Racket does not provide a typed version of **keyword-apply**, and it is unclear how to properly type it.

In conclusion, Racket supports keyword and optional arguments in a unique way via its powerful macro system. However, the support for first-class keyword arguments is very limited and cannot easily transfer to a type-safe setting.

5.4 Haskell

Unlike the aforementioned languages, Haskell does not support named arguments natively. However, the paradigm of *named arguments as records* has long existed in the Haskell community. Although we have to uncurry a function to have all parameters labeled in a record, it is clearer and more human-readable, especially when different parameters have the same type. For example, in the web server library *warp* [27], various server settings are bundled in the data type `Settings`, as shown in Fig. 7a. It is obvious how named arguments correspond to record fields, but it needs some thought on how to encode default values for optional arguments. The simplest approach, also used by *warp*, is to define a record `defaultSettings`, as shown in Fig. 7b. Users can update whatever fields they want to change while keeping others. For example, we update `settingsPort` and `settingsHost` while keeping the rest unchanged in Fig. 7c. Finally, we call the library function `runSettings` with the updated `settings` to run a server.

Such an approach works fine here but still has two drawbacks. The first issue is the dependency on `defaultSettings`. It is awkward for users to look for a record containing particular default values, especially when there are a

¹ `'` is quote, `‘` is quasiquote, and `,` is unquote in Racket.

<pre>data Settings = Settings { settingsPort :: Port , settingsHost :: HostPreference , settingsTimeOut :: Int , ... }</pre>	<pre>defaultSettings = Settings { settingsPort = 3000 , settingsHost = "*4" , settingsTimeout = 30 , ... }</pre>

(a) Record type.

(b) Default values.

```
runSettings :: Settings → Application → IO ()
runSettings = ...
```

```
main :: IO ()
main = runSettings settings app
  where settings = defaultSettings { settingsPort = 4000
                                    , settingsHost = "*6" }
```

(c) Updating some settings before running a server application.

Fig. 7: Named arguments as records in Haskell.

few similar records in a library. A better solution is to change the parameter of `runSettings` from a complete `Settings` to a function that updates `Settings`:

```
runSettings' :: (Settings → Settings) → Application → IO ()
runSettings' update = runSettings (update defaultSettings)
```

```
main :: IO ()
main = runSettings' update app
  where update settings = settings { settingsPort = 4000
                                    , settingsHost = "*6" }
```

With the new interface, users do not need to look for default values anymore. However, this design still has a second drawback: all arguments must have default values. Usually, we do not consider every argument to be optional. For example, we may want to require users to fill in `settingsPort`. A workaround employed by `SqlBackend` in the library *persistent* [21] is to have another function that asks for required arguments and supplements default values for optional arguments:

```
{-# language DuplicateRecordFields, RecordWildCards #-}
```

```
data ReqSettings = ReqSettings { settingsPort :: Port }
```

```
mkSettings :: ReqSettings → Settings
mkSettings ReqSettings {..} =
  Settings { settingsHost = "*4", settingsTimeout = 30, .. }
```

Best Practice. Although `mkSettings` resolves the second issue, there is a regression concerning the first issue: users have to look for `mk*` functions now. Fortunately, we can harmonize both design patterns to develop a third approach:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data OptSettings = OptSettings { settingsHost :: HostPreference
                                , settingsTimeout :: Int }

runSettings'' :: (OptSettings → Settings) → Application → IO ()
runSettings'' update = runSettings (update defaultSettings)
  where defaultSettings = OptSettings { settingsHost = "*4"
                                        , settingsTimeout = 30 }

main :: IO ()
main = runSettings'' update app
  where update OptSettings {..} =
        Settings { settingsPort = 4000, settingsHost = "*6", .. }
```

This last approach is probably the best practice in Haskell, though it is already quite complicated and requires two GHC language extensions. Of course, there could be other approaches to encoding named and optional arguments in Haskell. Users could get confused about the various available design patterns. This is largely due to lack of language-level support. We believe it is better for a language to natively support named and optional arguments.

Sidenote. The design pattern of *named arguments as records* can also be found in other functional languages like Standard ML, Elm, and PureScript, just to name a few. It is worth mentioning that these languages have first-class support for record types, so no separate type declarations are needed like in [Fig. 7a](#). However, they still suffer from lack of native support for optional arguments.

5.5 Record Calculi

Ohuri discussed how to model optional arguments in the future work of his seminal paper on compiling a polymorphic record calculus [19]. He proposed to extend a record calculus with optional-field selection ($e.\ell \ ? \ d$) which behaves like $e.\ell$ if ℓ is present in the record e or evaluates to d otherwise. However, his proposal is subject to a similar type-safety issue as `mypy`. The static type of e can easily lose track of the optional field ℓ and fail to ensure that $e.\ell$ has the same type as d at run time. Since Ohuri did not explicitly mention how to type-check optional-field selection, we cannot make any firm conclusion about the type safety of his proposal.

Osinski also discussed the support for optional arguments in Section 3.5 of his dissertation on compiling record concatenation [20]. His approach is based on row polymorphism and makes use of a sort of predicate on rows: $row_1 \blacktriangleright row_2$, which means that row_1 consists of all the fields in row_2 . With this predicate, a function

has type $\forall \rho. \text{row}_o \triangleright \rho \Rightarrow \{\text{row}_r, \rho\} \rightarrow \tau$ if the required and optional arguments are denoted by row_r and row_o , respectively. Roughly speaking, it means the parameter has a type between $\{\text{row}_r\}$ and $\{\text{row}_r, \text{row}_o\}$. At the term level, he introduced a compatible concatenation operator $|\&|$, which allows overlapping fields with the same types and prefers the fields on the right-hand side when overlapping occurs. An example of their translation is as follows:

```
fun add { x, y = 0 } = ...
(* is translated to *)
fun add r = let r' = { y = 0 } |&| r in
            let x = r'.x in let y = r'.y in ...
```

His approach is free from the type-safety issue, though based on a more sophisticated row-polymorphic system. There are two sorts of predicates and three variants of record concatenation operators in his calculus, for example, demonstrating some sophistication of his calculus.

It is worth noting that neither Ohori's nor Osinski's calculus supports subtyping. This is a significant limitation since subtyping is a common feature in many popular languages, especially object-oriented languages.

6 Conclusion

The benefits from named arguments are twofold. On the one hand, argument keywords serve as extra documentation at the language level. On the other hand, they lay the foundation for supporting commutativity and optionality.

Named and optional arguments are widely supported in mainstream programming languages but are hardly formalized. Our approach is inspired by existing mechanisms in OCaml and Haskell but further considers the interaction between first-class named arguments and subtyping. Static type checkers for Python and Ruby both suffer from a type-safety issue in this regard. We show that λ_{iu} can serve as a type-safe core calculus with compact support for intersection and union types.

We hope that this paper will call more attention to the foundation for named and optional arguments and inspire future language developers to consider potential type-safety issues more carefully in their designs.

Future Work. A natural extension of our work is to support omitting keywords if the order of arguments is not changed from how they are defined. In other words, this is a version without the property of distinctness as classified in Table 1. The syntactic approach employed by Scala can be a good reference. Another interesting direction is to evaluate our approach in a more practical setting, for example, by typing popular libraries like TensorFlow, where named and optional arguments are ubiquitous, and dependent default values can help simplify the functions and avoid some ill-defined parameter combinations.

Data Availability. The Coq formalization for this paper is available [30].

References

1. Aït-Kaci, H., Garrigue, J.: Label-selective λ -calculus: syntax and confluence. *Theor. Comput. Sci.* **151**(2) (1995). [https://doi.org/10.1016/0304-3975\(95\)00072-5](https://doi.org/10.1016/0304-3975(95)00072-5)
2. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Intersection and union types: syntax and semantics. *Inf. Comput.* **119**(2) (1995). <https://doi.org/10.1006/inco.1995.1086>
3. Castagna, G.: Programming with union, intersection, and negation types. In: *The French School of Programming*, chap. 12 (2023). https://doi.org/10.1007/978-3-031-34518-0_12
4. Charguéraud, A.: The locally nameless representation. *J. Autom. Reason.* **49**(3) (2012). <https://doi.org/10.1007/s10817-011-9225-2>
5. Church, A.: *The Calculi of Lambda-Conversion*. No. 6 in *Annals of Mathematics Studies*, Princeton University Press (1941), <https://press.princeton.edu/books/paperback/9780691083940/the-calculi-of-lambda-conversion>
6. Dunfield, J.: Elaborating intersection and union types. *J. Funct. Program.* **24**(2–3) (2014). <https://doi.org/10.1017/S0956796813000270>
7. Flatt, M., Barzilay, E.: Keyword and optional arguments in PLT Scheme. In: *Scheme* (2009), <https://users.cs.utah.edu/plt/publications/scheme09-fb.pdf>
8. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* **55**(4) (2008). <https://doi.org/10.1145/1391289.1391293>
9. Furuse, J.P., Garrigue, J.: A label-selective lambda-calculus with optional arguments and its compilation method. Tech. rep., Kyoto University (1995), <https://www.math.nagoya-u.ac.jp/~garrigue/papers/rims-1041.pdf>
10. Garrigue, J.: Objective Label trilogy, <http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/>
11. Garrigue, J.: Labeled and optional arguments for Objective Caml. In: *JSSST SIG-PPL* (2001), <https://www.math.nagoya-u.ac.jp/~garrigue/papers/ppl2001.pdf>
12. Garrigue, J., Aït-Kaci, H.: The typed polymorphic label-selective lambda-calculus. In: *POPL* (1994). <https://doi.org/10.1145/174675.174434>
13. Lehtosalo, J., et al.: Mypy: Optional static typing for Python, <https://www.mypy-lang.org>
14. Luo, Z., Soloviev, S., Xue, T.: Coercive subtyping: Theory and implementation. *Inf. Comput.* **223** (2013). <https://doi.org/10.1016/j.ic.2012.10.020>
15. Madhavapeddy, A., Minsky, Y.: *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press (2022). <https://doi.org/10.1017/9781009129220>
16. Matsumoto, S., et al.: Steep: Gradual typing for Ruby, <https://github.com/soutaro/steep>
17. Microsoft: TypeScript: JavaScript with syntax for types, <https://www.typescriptlang.org>
18. Nieto, A., Zhao, Y., Lhoták, O., Chang, A., Pu, J.: Scala with explicit nulls. In: *ECOOP* (2020). <https://doi.org/10.4230/LIPIcs.ECOOP.2020.25>
19. Ohori, A.: A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.* **17**(6) (1995). <https://doi.org/10.1145/218570.218572>

20. Osinski, E.: A polymorphic type system and compilation scheme for record concatenation. Ph.D. thesis, New York University (2006), https://cs.nyu.edu/media/publications/osinski_edward.pdf
21. Parsons, M., et al.: Persistent: Type-safe, multi-backend data serialization, <https://hackage.haskell.org/package/persistent>
22. Pierce, B.C., et al.: Programming language foundations. In: Software Foundations. <https://softwarefoundations.cis.upenn.edu/plf-current/>
23. Rehman, B.: A blend of intersection types and union types. Ph.D. thesis, The University of Hong Kong (2023), <https://hub.hku.hk/handle/10722/335090>
24. Reynolds, J.C.: Design of the programming language FORSYTHE. In: Algol-like Languages, vol. 1, chap. 8 (1997). https://doi.org/10.1007/978-1-4612-4118-8_9
25. Roblox: Luau: A fast, small, safe, gradually typed embeddable scripting language derived from Lua, <https://luau.org>
26. Rytz, L., Odersky, M.: Named and default arguments for polymorphic object-oriented languages: A discussion on the design implemented in the Scala language. In: SAC (2010). <https://doi.org/10.1145/1774088.1774529>
27. Snoyman, M., et al.: Warp: A fast, light-weight web server for WAI applications, <https://hackage.haskell.org/package/warp>
28. Stripe: Sorbet: A static type checker for Ruby, <https://sorbet.org>
29. Sun, Y., Dhandhanian, U., Oliveira, B.C.d.S.: Compositional embeddings of domain-specific languages. In: OOPSLA (2022). <https://doi.org/10.1145/3563294>
30. Sun, Y., Oliveira, B.C.d.S.: Named arguments as intersections, optional arguments as unions (artifact). Zenodo (2025). <https://doi.org/10.5281/zenodo.14697184>
31. Tobin-Hochstadt, S., St-Amour, V., Dobson, E., Takikawa, A.: The Typed Racket Guide, <https://docs.racket-lang.org/ts-guide/>
32. van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., Fisker, R.G.: Revised report on the algorithmic language ALGOL 68. *Acta Informatica* **5**(1–3) (1975). <https://doi.org/10.1007/BF00265077>
33. Zhang, W., Sun, Y., Oliveira, B.C.d.S.: Compositional programming. *ACM Trans. Program. Lang. Syst.* **43**(3) (2021). <https://doi.org/10.1145/3460228>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Context-Sensitive Demand-Driven Control-Flow Analysis

Tim Whiting^(✉) and Kimball Germane^{}

Brigham Young University, Provo UT 84601, USA

tim@whitings.org

kimball@cs.byu.edu

Abstract. By decoupling and decomposing control flows, demand control-flow analysis (CFA) resolves only the flow segments determined necessary to produce a specified control-flow fact. It therefore presents a more flexible interface and pricing model than typical CFA, making many useful applications practical. At present, the only realization of demand CFA is the context-insensitive Demand OCFA. Typical mechanisms for adding context sensitivity are not compatible with the demand setting because the analyzer is dispatched at arbitrary program points in indeterminate contexts. We overcome this challenge by identifying a context suitable for a demand analysis and designing a representation thereof that allows it to model incomplete knowledge of the context. On top of this design, we construct Demand *m*-CFA, a context-sensitive demand CFA hierarchy. With the attractive pricing model of demand analysis and the precision offered by context sensitivity, we show that Demand *m*-CFA can replace its exhaustive counterpart in compiler backends and integrate into interactive tools such as language servers.

Keywords: Demand CFA, *m*-CFA, Context-Sensitivity, Control Flow Analysis

1 Getting into the Flow

Conventional control-flow analysis is tactless—unthinking and inconsiderate. To illustrate, consider the program fragment below which defines a recursive `fold` function. As the function iterates, it updates the index `n` and accumulator `a` using the functions `f` and `g` respectively. The values of `f` and `g` flow in parallel within `fold`, each (1) being bound in the initial call, (2) flowing to its corresponding parameter, and (3) being directly called once per iteration. But the flows of `f` and `g` don't completely overlap: `f`'s value's flow begins at `sub1` and branches into the call to `g` whereas `g`'s value's comes from a reference to `h` and only is applied within `fold`.

```
(letrec ([fold
  (λ (f g n a)
    (if (zero? n)
      a
      (fold f g (f n) (g f n a))))])
  (fold sub1 h 42 1))
```

Now consider a tool focused on the call `(f n)` and seeking the value of `f` in order to, say, expand `f` inline. Only

the flow segments identified above respective to $\mathbf{f}$ are needed to fully determine this value—and know that it is fully-determined. Yet conventional control-flow analysis (CFA) is *exhaustive*, insistent on determining every segment of every flow, starting from the program’s entry point.¹ In the account it produces, the segmentation of individual flows and independence of distinct flows are completely implicit. To obtain $\mathbf{f}$ ’s value with a conventional CFA, the user must be willing to pay for $\mathbf{g}$ ’s—and any other values incidental to it—as well.

Inspired by demand dataflow analysis [4], a *demand* CFA does not determine every segment of every flow but only those segments which contribute to the values of specified program points. Moreover, because its segmentation of flows is explicit, it only need analyze each segment once and can reuse the results in any flow which contains the segment. In the `fold` example, a supporting demand CFA works backwards from the reference to $\mathbf{f}$ to determine its value, and considers only the three flow segments identified above to do so.

The interface and pricing model demand CFA offers make many useful applications practical. Horwitz et al. [8] identify several ways this practicality is realized:

1. One can restrict analysis to a program’s critical path or only where certain features are used.
2. One can analyze more often, and interleave analysis with other tools. For example, a demand analysis does not need to worry about optimizing transformations invalidating analysis results since one can simply re-analyze the transformed points.
3. One can let a user drive the analysis, even interactively, to enhance, e.g., an IDE experience. Having implemented our context-sensitive demand CFA in Visual Studio Code for a subset of the Koka language [10], our firsthand experience confirms its practicality here.

1.1 Adding Context Sensitivity to Demand CFA

Presently, the only realization of demand CFA is Demand 0CFA [5] which is context-insensitive. Adding context sensitivity to demand CFA would provide the same benefits that context sensitivity confers to analyses at large: increased precision and, in some cases, a reduced workload [14]. (In §2, we give an intuitive overview of Demand 0CFA and illustrate the positive effects of context sensitivity; in §4, we present a streamlined formulation of Demand 0CFA.)

However, the demand setting presents a challenge for adding context sensitivity: unlike exhaustive analyses in which the context is fully determined at each point in analysis, a demand analysis is deployed on an arbitrary program point in an undetermined context. Thus, the task of a context-sensitive demand CFA is not only to respect the context as far as it is known but also to determine

¹ Exhaustive CFA can be made to work with program components where free variables are treated specially (e.g. using Shivers’ escape technique [22, Ch. 3]). This special treatment does not change the fundamental *exhaustive* nature of the analysis nor bridge the shortcomings we describe.

unknown contexts as they are discovered relevant to analysis. We overcome this challenge through the confluence of (1) a compatible choice of context, (2) the representation of that context, and (3) that of environments, which we discuss in §5.

After addressing this challenge, we arrive at Demand *m*-CFA (§6), a hierarchy of context-sensitive demand CFA. Demand *m*-CFA determines the context only to the extent necessary to soundly answer analysis questions, as opposed to determining the entire context, which we illustrate in §2. This parsimony not only allows Demand *m*-CFA to avoid analysis work, it informs the analysis client which aspects of the context are relevant to the posed analysis question, which the client can feed into its formulation of subsequent questions.

Demand *m*-CFA is sound with respect to a concrete albeit demand semantics called *demand evaluation*, which is itself sound with respect to a standard call-by-value semantics.

We have implemented Demand *m*-CFA in several settings using the *Abstracting Definitional Interpreters* (ADI) technique [2]. We evaluate the implementation cost and performance to empirically assess Demand *m*-CFA (§8).

We conclude by discussing related (§9) and future work (§10).

2 Demand CFA, Intuitively

A user interacts with demand CFA by submitting queries, which the analyzer resolves online. There are two types of queries:

1. An *evaluation* query seeks the values to which a specified expression may evaluate. In essence, it is asking which values flow to a given expression.
2. A *trace* query seeks the sites at which the value of a specified expression may be used. In essence, it is asking where values flow from a given expression.

To resolve all but trivial queries, the analysis issues subqueries—of one type or the other—to resolve flows on which the original query depends.

We illustrate the operation of a demand CFA considering queries over the program

```
(let ([f (λ (x) x)]) (+ (f 42) (f 35)))
```

which is written in Pure Scheme (i.e. purely-functional Scheme).

2.1 Without Context Sensitivity

As many readers are likely unfamiliar with demand CFA, we'll first look at how Demand 0CFA, the context-*ins*sensitive embodiment of demand CFA, resolves queries.

Suppose that a user submits an evaluation query q_0 on the expression $(f\ 35)$. Since $(f\ 35)$ is a function application, Demand 0CFA issues a subquery q_1 to evaluate the operator f . For each procedure value of f , Demand 0CFA issues a subquery to determine the value of its body as the value of q_0 . To the left is a trace of the queries that follow q_0 .

```

 $q_0$  evaluate (f 35)
 $q_1$   evaluate f
 $q_2$   evaluate ( $\lambda$  (x) x)
       $\Rightarrow$  ( $\lambda$  (x) x)
 $q_3$  evaluate x
 $q_4$   trace ( $\lambda$  (x) x)
 $q_5$   trace f in (f 42)
       $\Rightarrow$  (f 42)
 $q_7$  evaluate 42
       $\Rightarrow$  42
 $q_6$   trace f in (f 35)
       $\Rightarrow$  (f 35)
 $q_8$  evaluate 35
       $\Rightarrow$  35

```

Indented queries denote subqueries whose results are used to continue resolution of the superquery. A subsequent query at the same indentation level is a query in “tail position”, whose results are those of a preceding query. A query often issues multiple queries in tail position, as this example demonstrates. The operator f is a reference, so Demand 0CFA walks the syntax to find where f is bound. Upon finding it bound by a `let` expression, Demand 0CFA issues a subquery q_2 to evaluate its bound expression $(\lambda (x) x)$. The expression $(\lambda (x) x)$ is a λ term—a value—which q_2 propagates directly to q_1 . Once q_1 receives it, Demand 0CFA issues a subquery q_3 for the evaluation of its body. Its body x is a reference, so Demand 0CFA walks the syntax to discover that it is λ -bound and therefore

that its value is the value of the argument at the application of $(\lambda (x) x)$. That this call to $(\lambda (x) x)$ originated at $(f 35)$ is contextual information, to which Demand 0CFA is insensitive. Consequently, Demand 0CFA issues a trace query q_4 to find all the application sites of $(\lambda (x) x)$. Because it is an expression bound to f , Demand 0CFA issues a subqueries q_5 and q_6 to find the use sites of both references to f . Each of these subqueries resolves immediately since each of the references is in operator position and their results are propagated to q_4 . For each result, q_3 issues a subquery— q_7 and q_8 —to evaluate the arguments, each of which is a numeric literal, whose value is immediately known. Each query propagates its results to q_3 which propagates them to q_0 which returns them to the user. Thus, Demand 0CFA concludes that $(f 35)$ may evaluate to 42 or 35.

2.2 With Context Sensitivity

We’ll now look at how Demand m -CFA, a context-sensitive demand CFA, resolves queries. As is typical, Demand m -CFA uses an environment to record the binding context of each in-scope variable. Hence, in this setting, queries and results include not only expressions, but environments as well. We will also see that Demand m -CFA does not need a timestamp to record the “current” context, a fact we discuss further in §5.5. Let’s consider the same evaluation query q_0 over $(f 35)$, this time in the top-level environment $\langle \rangle$. Like Demand 0CFA, Demand m -CFA issues the subquery q_1 to determine the operator f , also in $\langle \rangle$. After it discovers $(\lambda (x) x)$ to be the binding expression of f , it issues an evaluation query over it (q_2) again in $\langle \rangle$. The result of q_2 is $(\lambda (x) x)$ in $\langle \rangle$, essentially

```

 $q_0$  evaluate (f 35) in  $\langle \rangle$ 
 $q_1$   evaluate f in  $\langle \rangle$ 
 $q_2$   evaluate ( $\lambda$  (x) x) in  $\langle \rangle$ 
       $\Rightarrow$  ( $\lambda$  (x) x) in  $\langle \rangle$ 
 $q_3$  evaluate x in  $\langle (f 35) \rangle$ 
 $q'_3$  caller x in  $\langle (f 35) \rangle$ 
 $q_4$   trace ( $\lambda$  (x) x) in  $\langle \rangle$ 
 $q_5$   trace f in  $\langle \rangle$ 
       $\Rightarrow$  (f 42) in  $\langle \rangle$ 
       $\Rightarrow$  fail
 $q_6$   trace f in  $\langle \rangle$ 
       $\Rightarrow$  (f 35) in  $\langle \rangle$ 
       $\Rightarrow$  (f 35) in  $\langle \rangle$ 
 $q_8$  evaluate 35 in  $\langle \rangle$ 
       $\Rightarrow$  35 in  $\langle \rangle$ 

```

a closure. As before, this result is passed first to q_1 and then to q_0 at which point Demand m -CFA constructs q_3 , an evaluation query over its body. The query's environment $\langle (\mathbf{f} \ 35) \rangle$ records the context in which the parameter $\mathbf{x}$ was bound. In order to evaluate $\mathbf{x}$, Demand m -CFA issues a *caller* query q'_3 to determine the caller of $(\lambda (\mathbf{x}) \ \mathbf{x})$ that yielded the environment $\langle (\mathbf{f} \ 35) \rangle$. It then issues the trace query q_4 , this time subordinate to q'_3 , which issues q_5 and q_6 and results in the same two applications of $\mathbf{f}$. However, when q'_3 receives a caller from q_4 , Demand m -CFA ensures that the caller could produce the binding context of the parameter in q'_3 's environment. If so, q'_3 yields the result to q_3 ; if not, it cuts off the resolution process for that path. In this case, q_5 's result $(\mathbf{f} \ 42)$ is not compatible with q'_3 , and Demand m -CFA ceases resolving it rather than issuing q_7 . However, q_6 's result $(\mathbf{f} \ 35)$ is compatible, and its resolution continues, issuing q_8 . Resolution of q_8 occurs immediately and its result is propagated to the top-level query.

This example illustrates how Demand m -CFA uses the context information recorded in the environment to filter out discovered callers of a particular closure. Not only does this filtering increase precision in the expected way, but, in this example, it also prevents Demand m -CFA from issuing a spurious query (q_7). This behavior is an example of the well-known phenomenon of high precision keeping the analyzer's search space small [14].

2.3 ...And Indeterminacy

Each environment in the previous section was fully determined. Typically, Demand m -CFA resolves queries and produces results with environments that are—at least partially—indeterminate. For instance, to obtain all of the values

q_0 evaluate $\mathbf{x}$ in $\langle ? \rangle$ q'_0 caller $\mathbf{x}$ in $\langle ? \rangle$ q_1 trace $(\lambda (\mathbf{x}) \ \mathbf{x})$ in $\langle \rangle$ q_2 trace $\mathbf{f}$ in $(\mathbf{f} \ 42)$ in $\langle \rangle$ $\Rightarrow (\mathbf{f} \ 42)$ in $\langle \rangle$ $\Rightarrow (\mathbf{f} \ 42)$ in $\langle \rangle$ q_4 evaluate $\mathbf{x}$ in $\langle (\mathbf{f} \ 42) \rangle$ q_5 evaluate 42 in $\langle \rangle$ $\Rightarrow 42$ in $\langle \rangle$ q_3 trace $\mathbf{f}$ in $(\mathbf{f} \ 35)$ in $\langle \rangle$ $\Rightarrow (\mathbf{f} \ 35)$ in $\langle \rangle$ q_6 evaluate $\mathbf{x}$ in $\langle (\mathbf{f} \ 35) \rangle$ q_7 evaluate 35 in $\langle \rangle$ $\Rightarrow 35$ in $\langle \rangle$	to which $\mathbf{x}$, the body of $(\lambda (\mathbf{x}) \ \mathbf{x})$, may evaluate, a user may issue the query evaluate $\mathbf{x}$ in $\langle ? \rangle$ (q_0 at left) where $?$ is a “wildcard” context to be instantiated with each context the analyzer discovers. (Though each context in the environment is indeterminate, the shape of the environment itself is determined by the lexical binding structure, which we discuss further in §5.3.) Once issued, resolution of $\mathbf{x}$'s evaluation again depends on a caller query q'_0 . However, because the parameter $\mathbf{x}$'s context is unknown, rather than filtering out callers, the caller query causes $?$ to be instantiated with a context derived from each caller. As before, Demand m -CFA dispatches a trace query q_1 which then

traces occurrences of $\mathbf{f}$ via q_2 and q_3 . This query locates the call sites $(\mathbf{f} \ 42)$ in $\langle \rangle$ and $(\mathbf{f} \ 35)$ in $\langle \rangle$. Once q_2 delivers the result $(\mathbf{f} \ 42)$ in $\langle \rangle$ to q_1 and then q'_0 , Demand m -CFA *instantiates* q_0 with this newly-discovered caller to form q_4 , whose result is q_0 's also. After creating q_3 , it continues with its resolution by

issuing q_4 to evaluate the argument 42 in $\langle \rangle$. Its result of 42 propagates from q_4 to q_0 ; The instantiation from q_3 proceeds similarly.

3 Language and Notation

We present Demand m -CFA over the unary λ calculus. It is straightforward to extend it to multiple-arity functions, data structures, constants, primitives, conditionals, and recursive forms—which we have in our implementations—but this small language suffices to discuss the essential aspects of context sensitivity.

In order to keep otherwise-identical expressions distinct, many CFA presentations uniquely label program sub-expressions.² This approach would be used, for example, to disambiguate the two references to $\mathbf{f}$ in the program in §2. We instead use the syntactic context of each expression, which uniquely determines it, as a de-facto label, since Demand CFA consults it extensively.

In the unary λ calculus, expressions e adhere to the grammar on the left and syntactic contexts C adhere to the grammar on the right.

$$\text{Exp} \ni e ::= x \mid \lambda x.e \mid (ee) \quad \text{SynCtx} \ni C ::= \lambda x.C \mid (Ce) \mid (eC) \mid \square.$$

The syntactic context C of an instance of an expression e within a program pr is pr itself with a hole $\square$ in place of the selected instance of e . For example, the program $\lambda x.(xx)$ contains two references to x , one with syntactic context $\lambda x.(\square x)$ and the other with $\lambda x.(x\square)$.

The composition $C[e]$ of a syntactic context C with an expression e consists of C with $\square$ replaced by e , so that $C[e]$ denotes the program with a focus on e . For example, we focus on the reference to x in operator position in the program $\lambda x.(xx)$ with $\lambda x.([x]x)$. To aid visual parsing, we also shade the context to distinguish it from the focused expression.

The composition of a context and expression is a cursor, drawn from the space $\text{Cursor} := \text{Exp} \times \text{SynCtx}$. In a cursor, we typically leave the context unspecified, referring to, e.g., a reference to x by $C[x]$ and two distinct references to x by $C_0[x]$ and $C_1[x]$ (where $C_0 \neq C_1$). The immediate syntactic context of an expression is often relevant, however, and we make it explicit by a double composition $C_0[C_1[e]]$. For example, we use $C[[x]x]$ to focus on the expression x in the operator context $(\square x)$ in the context C .

4 Demand 0CFA

Demand 0CFA has two modes of operation, *evaluation* and *tracing*, which users access by submitting evaluation and trace queries, respectively. An evaluation query resolves the values to which a designated expression may evaluate and a trace query resolves the sites which may apply the value of the designated

² Others operate over a form which itself names all intermediate results, such as CPS or $\mathcal{A}$ -normal form, and identify each expression by its associated (unique) name.

expression. These modes are essentially dual and reflect the dual perspective of exhaustive CFA as an account of either (1) the λ terms $\lambda x.e$ which may be applied at each application site $(e_0 e_1)$, or (2) the application sites $(e_0 e_1)$ at which each λ term $\lambda x.e$ may be applied. However, in contrast to exhaustive CFA, Demand 0CFA resolves evaluation queries over individual expressions with a straightforward way to share work across the resolution of related queries. (It also resolves trace queries over individual expressions, but exhaustive CFAs have no counterpart to this functionality.)

The evaluation and trace modes of operation are effected by the big-step relations $\Downarrow_{eval}, \Rightarrow_{expr} \subseteq \mathbf{Cursor} \times \mathbf{Cursor}$, which are defined mutually inductively. These relations are respectively supported by the auxiliary metafunction $\mathbf{bind} \in \mathbf{Var} \times \mathbf{Cursor} \rightarrow \mathbf{Cursor}$ and relation $\Rightarrow_{find} \subseteq \mathbf{Var} \times \mathbf{Cursor} \times \mathbf{Cursor}$, which connect references to bindings and vice versa. Anticipating the addition of context sensitivity, we define $\Downarrow_{eval}$ in terms of the relation $\Downarrow_{call} \subseteq \mathbf{Cursor} \times \mathbf{Cursor}$, which we discuss below. Figure 1 presents the definitions of all of these relations. The judg-

$$\begin{array}{c}
\Downarrow_{eval}, \Rightarrow_{expr}, \Downarrow_{call} \subseteq \mathbf{Cursor} \times \mathbf{Cursor} \qquad \Rightarrow_{find} \subseteq \mathbf{Var} \times \mathbf{Cursor} \times \mathbf{Cursor} \\
\\
\text{LAM} \qquad \frac{}{C[\lambda x.e] \Downarrow_{eval} C[\lambda x.e]} \qquad \text{APP} \qquad \frac{C'[(e_0 e_1)] \Downarrow_{eval} C'[\lambda x.e] \quad C'[\lambda x.[e]] \Downarrow_{eval} C_v[\lambda x.e_v]}{C'[(e_0 e_1)] \Downarrow_{eval} C_v[\lambda x.e_v]} \\
\\
\text{REF} \qquad \frac{C'[e_x] = \mathbf{bind}(x, C[x]) \quad C'[e_x] \Downarrow_{call} C''[(e_0 e_1)] \quad C''[(e_0 [e_1])] \Downarrow_{eval} C_v[\lambda x.e]}{C[x] \Downarrow_{eval} C_v[\lambda x.e]} \\
\\
\text{OPERATOR} \qquad \frac{}{C[(e_0 e_1)] \Rightarrow_{expr} C[(e_0 e_1)]} \\
\\
\text{BODY} \qquad \frac{C[\lambda x.[e]] \Downarrow_{call} C'[(e_0 e_1)] \quad C'[(e_0 e_1)] \Rightarrow_{expr} C''[(e_2 e_3)]}{C[\lambda x.[e]] \Rightarrow_{expr} C''[(e_2 e_3)]} \\
\\
\text{OPERAND} \qquad \frac{C'[(e_0 e_1)] \Downarrow_{eval} C'[\lambda x.e] \quad x, C'[\lambda x.[e]] \Rightarrow_{find} C_x[x] \quad C_x[x] \Rightarrow_{expr} C'[(e_2 e_3)]}{C[(e_0 [e_1])] \Rightarrow_{expr} C'[(e_2 e_3)]} \\
\\
\text{CALL} \qquad \frac{C[\lambda x.e] \Rightarrow_{expr} C'[(e_0 e_1)]}{C[\lambda x.[e]] \Downarrow_{call} C'[(e_0 e_1)]} \qquad \text{FIND-REF} \qquad \frac{}{x, C[x] \Rightarrow_{find} C[x]} \qquad \text{FIND-OPERATOR} \qquad \frac{}{x, C[(e_0 e_1)] \Rightarrow_{find} C_x[x]} \\
\\
\text{FIND-OPERAND} \qquad \frac{x, C[(e_0 [e_1])] \Rightarrow_{find} C_x[x]}{x, C[(e_0 e_1)] \Rightarrow_{find} C_x[x]} \qquad \text{FIND-BODY} \qquad \frac{x \neq y \quad x, C[\lambda y.[e]] \Rightarrow_{find} C_x[x]}{x, C[\lambda y.e] \Rightarrow_{find} C_x[x]}
\end{array}$$

Fig. 1: Demand 0CFA evaluation and trace relations

ment $C[e] \Downarrow_{eval} C_v[\lambda x.e_v]$ denotes that the expression e (residing in syntactic

context C) evaluates to (a closure over) $\lambda x.e_v$. (In a context-insensitive analysis, we may represent a closure by the λ term itself.) Demand 0CFA arrives at such a judgment, as an interpreter does, by considering the type of expression being evaluated. The *Lam* rule captures the intuition that a λ term immediately evaluates to itself. The *App* rule captures the intuition that an application evaluates to whatever the body of its operator does. Hence, if the operator e_0 evaluates to $\lambda x.e$, and e evaluates to $\lambda y.e_v$, then the application $(e_0 e_1)$ evaluates to $\lambda y.e_v$ as well. Notice that the *App* does not evaluate the argument; if the argument is needed, indicated by a reference to the operator's parameter x during evaluation of its body, the *Ref* rule obtains it. The *Ref* rule captures the intuition that a reference to a parameter x takes on the value of the argument at each call site where the λ which binds x is called. The *bind* metafunction determines the binding configuration of x by walking outward on the syntax tree until it encounters x 's binder. Figure 2 presents its definition, and we note the absence of a rule for the case where x is unbound, since we define programs as closed expressions.

$\text{bind} : \text{Var} \times \text{Cursor} \rightarrow \text{Cursor}$

$$\begin{aligned} \text{bind}(x, C[(e_0) e_1]) &= \text{bind}(x, C[(e_0 e_1)]) \\ \text{bind}(x, C[(e_0 [e_1])]) &= \text{bind}(x, C[(e_0 e_1)]) \\ \text{bind}(x, C[\lambda y.[e]]) &= \text{bind}(x, C[\lambda y.e]) \text{ where } y \neq x \\ \text{bind}(x, C[\lambda x.[e]]) &= C[\lambda x.[e]] \end{aligned}$$

Fig. 2: The *bind* metafunction

A judgment $C[\lambda x.[e]] \Downarrow_{\text{call}} C'[(e_0 e_1)]$ denotes that the application $(e_0 e_1)$ applies $\lambda x.e$, thereby binding x . Demand 0CFA arrives at this judgment by the *Call* rule which uses the $\Rightarrow_{\text{expr}}$ relation to determine it. In Demand 0CFA, this relation is only a thin wrapper over $\Rightarrow_{\text{expr}}$, but becomes more involved with the addition of context sensitivity.

A judgment $C[e] \Rightarrow_{\text{expr}} C'[(e_0 e_1)]$ denotes that the value of the expression e is applied at $(e_0 e_1)$. Demand 0CFA arrives at such a judgment by considering the type of the syntactic context to which the value flows. The *Operator* rule captures the intuition that, if $\lambda x.e$ flows to operator position e_0 of $(e_0 e_1)$, it is applied by $(e_0 e_1)$. The *Body* rule captures the intuition that if a value flows to the body of a λ term, then it flows to each of its callers as well. The *Operand* rule captures the intuition that a value in operand position is bound by the formal parameter of each operator value and hence to each reference to the formal parameter in the operator's body. If the operator e_f evaluates to $\lambda x.e$, then the value of e_a flows to each reference to x in e .

The $\Rightarrow_{\text{find}}$ relation associates a variable x and expression e with each reference to x in e . Its purpose is to determine where values flow locally within an expression by using and enforcing lexical scope. *Find-Ref* finds e itself if e is a reference to x . *Find-Operator* and *Find-Operand* find references to x in $(e_0 e_1)$ by searching the operator e_0 and operand e_1 , respectively. *Find-Body* finds ref-

erences to x in $\lambda x.e$ taking care that $x \neq y$ so that it does not find shadowed references.

5 Adding Context Sensitivity

A context-*insensitive* CFA is characterized by each program variable having a single entry in the store, shared by all bindings to it. In contrast, a context-sensitive CFA considers the context in which each variable is bound and ensures that only bindings made in the same context share a store entry. By extension, a context-sensitive CFA evaluates an expression under a particular environment, which identifies the context in which free variables within the expression are bound. (Like Demand 0CFA, our context-sensitive demand CFA does not materialize the store in the semantics, but it can be recovered if desired.)

A key constraint in the demand setting is that queries may be (and typically are) issued with incomplete context information. The analysis is expected both to determine the context to the extent necessary to resolve the query (but no more) and also respect the context so that the analysis is in fact context-sensitive. One implication of this expectation is that queries over expressions which are evaluated in multiple contexts should qualify results by the contexts in which they occur.

To achieve context-sensitive CFA, we must overcome three challenges: (1) identify a notion of context compatible with the demand CFA, (2) determine a representation of that context which can account for fully- or partially-indeterminate contexts, and (3) determine an appropriate representation of environments which record the context. We meet each challenge in this section.

5.1 Challenge 1: An Appropriate Notion of Context

To formulate context-sensitive demand CFA in the most general setting possible, we will avoid sensitivities to properties not present in our semantics, such as types. Since we focus on an untyped λ calculus, the most straightforward choice is call-site sensitivity.

The canonical call-site sensitivity is that of k -CFA [22] which sensitizes each binding to the last k call sites encountered in evaluation. However, this form of call-site sensitivity works against the parsimonious nature of demand CFA. To make this concrete, consider that, in the fragment `(begin (f x) (g y))`, the binding of g 's parameter under a last- k -call-site sensitivity depends on the particular calls made during the evaluation of `(f x)`. If the value of `(f x)` is not otherwise demanded, this dependence either provokes demand analysis to discover more of the context or requires that the portion of the context contributed by `(f x)` be left indeterminate, thereby sacrificing precision.

A better form of call-site sensitivity would allow demand CFA to discover the context through its natural course of operation. Such a form, it turns out, is the top- m -stack-frames sensitivity offered by m -CFA.

***m*-CFA’s context abstraction** The *m*-CFA hierarchy [15] is the result of an investigation into the polynomial character of *k*-CFA in an object-oriented setting versus its exponential character in a functional setting. The crucial discovery of that investigation was that object-oriented (OO) *k*-CFA induces flat environments whereas functional-oriented *k*-CFA induces nested environments. Specifically, OO *k*-CFA re-binds object (closure) variables in the allocation context which collapses the exponential environment space to a polynomial size. From this insight they derive *m*-CFA, which simulates the OO binding discipline even when analyzing functional programs; that is, when applied, *m*-CFA binds a closure’s arguments and rebinds its environment’s bindings all in the binding context of the application. Under this policy, within a given environment, all bindings are in the same context and, consequently, the analysis can represent that environment simply as that binding context.

However, this binding policy amplifies a weakness of the *k*-most-recent-call-sites abstraction of *k*-CFA. Consider a [$k = 2$]CFA analysis of a call to **f** defined by

```
(define (f x) (log "called f") (g x))
```

and suppose that **log** itself does not make any calls. When control reaches **(g x)**, the most-recent call is always **(log "called f")** so the binding context of **g**’s parameter, the most-recent two calls **(log "called f")** and **(g x)**, determine only one point in the program—the body of **f**. In *k*-CFA, only **g**’s parameter binding suffers this abbreviated precision; the bindings in **g**’s closure environment refer to the context in which they were introduced into it. In *m*-CFA, however, the bindings of **g**’s closure environment are re-bound in the context of **g**’s application as **g** is applied and once-distinct bindings are merged together in a semi-degenerate context.

To accommodate this binding policy, Might et al. [15] use the top-*m* stack frames as a binding context rather than the last *k* call sites as the former is unaffected by static sequences of calls. Hence, when control reaches **(g x)** in [$m = 2$]CFA analysis, the binding context is that of **f**’s entry and **g**’s parameter is bound in the context of **(g x)** and **f**’s caller—no context is wasted.

Because we’re using *m*-CFA’s top-*m*-stack-frames context abstraction, we call our context-sensitive demand CFA *Demand m-CFA*. It is important to keep in mind, however, that we do *not* adopt its re-binding policy. This is due to the fact that re-binding is defined in terms of a central store, which demand CFA does not explicitly model.

5.2 Challenge 2: Representing (Indeterminate) Contexts

Now that we have identified a context abstraction, we must choose a representation for it which will allow us to model incomplete knowledge. As a fully-determined context in *m*-CFA is an *m*-length sequence of call sites, a straightforward choice would be an *m*-length vector of possibly-indeterminate call sites which Demand *m*-CFA could fill in as it discovers them. However, this representation fails to capture the useful invariant that call sites within a context are

always discovered in sequence from the first (representing the top frame on the stack) to the last.

To illustrate, suppose we are evaluating x in the function $(\lambda (x) x)$ and that we have no knowledge about the binding context of x . In order to determine x 's value, we must determine the sites that call $(\lambda (x) x)$. If our analyzer determines that one such site is $(f\ 42)$, then it learns that the *top* frame—the first of the top- m frames—is $(f\ 42)$. The next frame is whatever the top frame of $(f\ 42)$'s context is, which may be indeterminate. Thus, the analyzer's knowledge of the top- m frames always increases from the top down. (This invariant holds when the analyzer enters a call as well, since it necessarily knows the call site when doing so.)

We devise a context representation that captures this invariant. A context cc^m representing m stack frames is either completely indeterminate, a pair of a call $C[(e_0\ e_1)]$ and a context cc^{m-1} of $m-1$ frames, or the stack bottom $()$ (which occurs at the top level of evaluation). A context cc^0 of zero frames (which is a different notion than the stack bottom) is simply the degenerate $\square$. Formally, we have

$$Context_0 \ni cc^0 ::= \square \quad Context_m \ni cc^m ::= ? \mid (C[(e_0\ e_1)], cc^{m-1}) \mid ()$$

With the context representation chosen, we can now turn to the environment representation.

5.3 Challenge 3: Representing Environments

In a lexically-scoped language, the environment at the reference x in the fragment

```
(define f (λ (x y) ... (λ (z) ... (λ (w) ... x ...) ...) ...))
```

contains bindings for x , y , z , and w . Exhaustive CFAs typically model this environment as a finite map from variables to contexts (i.e., the type $\text{Var} \rightarrow \text{Context}$). We instead use a variable's de Bruijn index, which is statically determined by the program syntax, and model environments as a sequence Context^* which, for any finite program, has length bounded by the maximum lexical depth therein.

Given this environment representation, we make one final tweak to the definition of contexts: we will qualify an indeterminate context $?$ with the parameter of the function whose context it represents, and assume programs are α -converted—i.e. that all bound variables are unique.³ This way, even an environment of completely indeterminate contexts still determines the expression it closes. For instance, we represent the indeterminate environment of y in $(\lambda (x) ((\lambda (y) y) (\lambda (z) z)))$ by $\langle ?_y, ?_x \rangle$ which is distinct from the indeterminate environment of z , which we represent by $\langle ?_z, ?_x \rangle$, even though they have the same shape. As explained in the next section, this distinction is crucial when considering instantiating indeterminate contexts.

³ In practice, we use the syntactic context of the body instead of the parameter, which is unique even if the program is not α -converted.

5.4 Instantiating Contexts

Demand m -CFA, at certain points during resolution, discovers information about the context in which the resolved flow occurs, and must instantiate *relevant* environments with that information. For instance, in the program

```
(let ([apply (λ (f) (λ (x) (f x)))])
  (+ ((apply add1) 42)
     ((apply sub1) 35)))
```

suppose that Demand m -CFA is issued an evaluation query for $(f\ x)$ in the environment $\langle ?_x, ?_f \rangle$, i.e., the fully-indeterminate environment. With our global view, we can see that $(f\ x)$ evaluates to 43 and 34. Let's consider how Demand m -CFA would arrive at the same conclusion.

First, it would trace $(\lambda (f) (\lambda (x) (f\ x)))$ in the environment $\langle \rangle$ to the binding `apply` and then to the call sites $(\text{apply}\ \text{add1})$ in $\langle \rangle$ and $(\text{apply}\ \text{sub1})$ in $\langle \rangle$. Each of these sites provides a context in which f is bound, so the evaluation of $(f\ x)$ continues in two instantiations of $\langle ?_x, ?_f \rangle$: the first to $\langle ?_x, (\text{apply}\ \text{add1}) :: ?_{tl} \rangle$ and the second to $\langle ?_x, (\text{apply}\ \text{sub1}) :: ?_{tl} \rangle$, where $?_{tl}$ is a dummy variable for the top-level context (the stack bottom). To evaluate either `add1` or `sub1`, its argument is needed, which flows through x . Then the two callers of $(\lambda (x) (f\ x))$ must be resolved. When $(\lambda (x) (f\ x))$ flows to the result of $(\text{apply}\ \text{add1})$, it is applied immediately at $((\text{apply}\ \text{add1})\ 42)$. Similarly, when it flows to the result of $(\text{apply}\ \text{sub1})$, it is applied immediately at $((\text{apply}\ \text{sub1})\ 35)$. These two call sites provide the binding contexts for x .

We might be tempted at this point to blindly instantiate $?_x$ with each of these sites. However, in doing so, we will get *six* environments, instantiating $?_x$ in each of $\langle ?_x, ?_f \rangle$, $\langle ?_x, (\text{apply}\ \text{add1}) :: ?_{tl} \rangle$, and $\langle ?_x, (\text{apply}\ \text{sub1}) :: ?_{tl} \rangle$ with each of $((\text{apply}\ \text{add1})\ 42) :: ?_{tl}$ and $((\text{apply}\ \text{sub1})\ 35) :: ?_{tl}$. In particular, we obtain

$$\langle ((\text{apply}\ \text{sub1})\ 35) :: ?_{tl}, (\text{apply}\ \text{add1}) :: ?_{tl} \rangle \quad \langle ((\text{apply}\ \text{add1})\ 42) :: ?_{tl}, (\text{apply}\ \text{sub1}) :: ?_{tl} \rangle$$

which do not correspond to any environment which arises in an exhaustive analysis.

The issue is that blindly instantiating indeterminate contexts ignores the evaluation path taken to arrive at the call site. In particular, it ignores where the nesting λ is applied, which must be applied first (as we observed in the previous section). In this example, we must consider the context of $(\lambda (f) (\lambda (x) (f\ x)))$ before we instantiate the context of $(\lambda (x) (f\ x))$.

The solution, then, is to not substitute a indeterminate context with a more-determined context, but an entire environment headed by an indeterminate context with that same environment headed by the more-determined one. This policy would, in this example, lead to all occurrences of

$$\langle ?_x, (\text{apply}\ \text{add1}) :: ?_{tl} \rangle \quad \text{substituted with} \quad \langle ((\text{apply}\ \text{add1})\ 42) :: ?_{tl}, (\text{apply}\ \text{add1}) :: ?_{tl} \rangle$$

and

$$\langle ?_x, (\text{apply}\ \text{sub1}) :: ?_{tl} \rangle \quad \text{substituted with} \quad \langle ((\text{apply}\ \text{sub1})\ 35) :: ?_{tl}, (\text{apply}\ \text{sub1}) :: ?_{tl} \rangle$$

and no others, which is precisely what we would hope.

This policy is effective even when the result of the function does not depend on both values. For instance, when Demand m -CFA evaluates $\mathbf{x}$ in $(\lambda (\mathbf{f}) (\lambda (\mathbf{x}) \mathbf{x}))$, it must still determine the caller of $(\lambda (\mathbf{f}) (\lambda (\mathbf{x}) \mathbf{x}))$ to determine the downstream caller of $(\lambda (\mathbf{x}) \mathbf{x})$.

Crucially, we need the indeterminate contexts to be qualified by the parameter of the function whose context it represents as mentioned previously. Without this distinction, we could instantiate both $?_x$ and $?_f$ with contexts that are only statically possible for the other. This is a clear win in terms of precision and performance, and does not distinguish more contexts than necessary due to the fact that the indeterminate variables in closure environments can be uniquely derived from their lambda.

5.5 Whence the timestamp?

In addition to introducing environments, context-sensitivity also typically introduces “timestamps” which serve as snapshots of the context at each evaluation step. For instance, classic k -CFA evaluates *configurations*, consisting of an expression, environment, and timestamp, to results.

With a top- m -stack-frames abstraction, the “current context” is simply the context of the variable(s) most recently bound in the environment. Lexical scope makes identifying these variables straightforward and our environment representation makes accessing their binding context straightforward as well. Thus, under such an abstraction, the environment uniquely determines the timestamp, and we can omit timestamps from configurations with no loss of information.

With the context, its representation, and the environment’s representation identified, we are now ready to define Demand m -CFA.

6 Demand m -CFA

$$\begin{aligned}
 \text{bind} &: \text{Var} \times \text{Cursor} \times \text{Env} \rightarrow \text{Cursor} \times \text{Env} \\
 \text{bind}(x, C[[e_0] e_1], \hat{\rho}) &= \text{bind}(x, C[(e_0 e_1)], \hat{\rho}) \\
 \text{bind}(x, C[(e_0 [e_1])], \hat{\rho}) &= \text{bind}(x, C[(e_0 e_1)], \hat{\rho}) \\
 \text{bind}(x, C[\lambda y. [e]], \hat{c}c :: \hat{\rho}) &= \text{bind}(x, C[\lambda y. e], \hat{\rho}) \text{ where } y \neq x \\
 \text{bind}(x, C[\lambda x. [e]], \hat{\rho}) &= (C[\lambda x. [e]], \hat{\rho})
 \end{aligned}$$

Fig. 3: The bind metafunction

Demand m -CFA augments Demand 0CFA with environments and a mechanism to instantiate contexts within environments, which together provide context sensitivity. The addition of environments pervades $\Downarrow_{eval}^m$, $\Rightarrow_{expr}^m$, and $\Rightarrow_{find}^m$ which are otherwise identical to their Demand 0CFA counterparts; these enriched relations are presented in Figure 4. The mechanism is located in $\Downarrow_{call}^m$, which is significantly elaborated relative to its Demand 0CFA counterpart.

$$\begin{aligned}
& \Downarrow_{eval}^m, \Rightarrow_{expr}^m, \Downarrow_{call}^m \subseteq \mathbf{Cursor} \times \mathbf{Env} \times \mathbf{Cursor} \times \mathbf{Env} \\
& \Rightarrow_{find}^m \subseteq \mathbf{Var} \times \mathbf{Cursor} \times \mathbf{Env} \times \mathbf{Cursor} \times \mathbf{Env}
\end{aligned}$$

LAM $\frac{}{C[\lambda x.e], \hat{\rho} \Downarrow_{eval}^m C[\lambda x.e], \hat{\rho}}$	OPERATOR $\frac{}{C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{expr}^m C[(e_0 \ e_1)], \hat{\rho}}$
REF $\frac{(C_x[e_x], \hat{\rho}_x) = \mathbf{bind}(x, C[x], \hat{\rho}) \quad C'_x[(e_0 \ e_1)], \hat{\rho}' \Downarrow_{eval}^m C_v[\lambda x.e], \hat{\rho}_v}{C[x], \hat{\rho} \Downarrow_{eval}^m C_v[\lambda x.e], \hat{\rho}_v}$	
APP $\frac{C'[(e_0 \ e_1)], \hat{\rho} \Downarrow_{eval}^m C'[\lambda x.e], \hat{\rho}' \quad C'_x[\lambda x.[e]], \mathbf{succ}_m(C[(e_0 \ e_1)], \hat{\rho}) :: \hat{\rho}' \Downarrow_{eval}^m C_v[\lambda x.e_v], \hat{\rho}_v}{C[(e_0 \ e_1)], \hat{\rho} \Downarrow_{eval}^m C_v[\lambda x.e_v], \hat{\rho}_v}$	
OPERAND $\frac{x, C'_x[\lambda x.[e]], \mathbf{succ}_m(C[(e_0 \ e_1)], \hat{\rho}) :: \hat{\rho}' \Rightarrow_{find}^m C_x[x], \hat{\rho}_x \quad C_x[x], \hat{\rho}_x \Rightarrow_{expr}^m C''[(e_2 \ e_3)], \hat{\rho}''}{C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{expr}^m C''[(e_2 \ e_3)], \hat{\rho}''}$	
BODY $\frac{C[\lambda x.[e]], \hat{\rho} \Downarrow_{call}^m C'_x[(e_0 \ e_1)], \hat{\rho}' \quad C'_x[(e_0 \ e_1)], \hat{\rho}' \Rightarrow_{expr}^m C''[(e_2 \ e_3)], \hat{\rho}''}{C[\lambda x.[e]], \hat{\rho} \Rightarrow_{expr}^m C''[(e_2 \ e_3)], \hat{\rho}''}$	
FIND-REF $\frac{}{x, C[x], \hat{\rho} \Rightarrow_{find}^m C[x], \hat{\rho}}$	FIND-OPERATOR $\frac{}{x, C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x \quad x, C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x}$
FIND-OPERAND $\frac{x, C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x}{x, C[(e_0 \ e_1)], \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x}$	FIND-BODY $\frac{x \neq y \quad x, C[\lambda y.[e]], ?_y :: \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x}{x, C[\lambda y.e], \hat{\rho} \Rightarrow_{find}^m C_x[x], \hat{\rho}_x}$

Fig. 4: Demand m -CFA Resolution

In Demand m -CFA, environments are constructed when the analysis follows evaluation forward, such as entering a call, and instantiated when the analysis follows it backward, such as when the caller of an entry configuration is sought.

6.1 Following Evaluation Forwards

When a call is entered, which occurs in the *App* and *Operand* rules, a new environment is synthesized using the succ_m metafunction which determines the binding context of the call as

$$\text{succ}_m(C[(e_0 e_1)], \hat{c}c :: \hat{\rho}) = [C[(e_0 e_1)] :: \hat{c}c]_m$$

where $[\cdot]_m$ is defined

$$[\hat{c}c]_0 = \square \quad [?_x]_m = ?_x \quad [C[(e_0 e_1)] :: \hat{c}c]_m = C[(e_0 e_1)] :: [\hat{c}c]_{m-1}$$

6.2 Following Evaluation Backwards

When the value of a reference is demanded, Demand m -CFA first uses the *bind* metafunction to locate its binding configuration. Its definition, presented in Figure 3, is lifted from Demand 0CFA's to accommodate environments.

With the binding configuration in hand, Demand m -CFA issues a call query to resolve calls which enter that configuration. The $\Downarrow_{call}^m$ relation resolves such queries. As in Demand 0CFA, $\Downarrow_{call}^m$ defers to $\Rightarrow_{expr}^m$ to determine the call configurations at which a particular closure is applied. In the presence of contexts, however, there are two possibilities when a call configuration is resolved. The first possibility is that the call entry context computed from that configuration precisely matches that of the binding configuration, in which case the call configuration is a caller of the binding configuration; we refer to this as the *Known-Call* case. The second possibility is that the call entry context refines the binding configuration's context, in which case the refined context and environment should be instantiated to the refining context and environment; we refer to this as the *Unknown-Call* case.

6.3 Discovering Callers

We now describe how $\Downarrow_{call}^m$, presented in Figure 5, handles each of these cases. Each case is handled by a corresponding rule. Each rule is predicated on the *reachability* of the call query, which we discuss shortly, and resolution of the corresponding trace query. The *Known-Call* rule says that, if the call entry context of the delivered call matches the binding configuration's, then the delivered call is a known call—*known* because the caller query has the context of the call already in its environment. If $\hat{c}c_1 \neq \hat{c}c_0$, however, then the result constitutes an *unknown* caller. In this case, *Unknown-Call* considers whether $\hat{c}c_1$ refines $\hat{c}c_0$

$$\begin{array}{c}
\text{KNOWN-CALL} \\
\frac{C[\lambda x.e], \hat{\rho} \Rightarrow_{\text{expr}}^m C'[(e_0 \ e_1)], \hat{\rho}' \quad q \uparrow_{\text{reach}}^m \text{call}(C[\lambda x.[e]], \hat{c}c_0 :: \hat{\rho}) \quad \hat{c}c_1 := \text{succ}_m(C'[(e_0 \ e_1)], \hat{\rho}') \quad \hat{c}c_1 = \hat{c}c_0}{C[\lambda x.[e]], \hat{c}c_0 :: \hat{\rho} \Downarrow_{\text{call}}^m C'[(e_0 \ e_1)], \hat{\rho}'} \\
\\
\text{UNKNOWN-CALL} \\
\frac{C[\lambda x.e], \hat{\rho} \Rightarrow_{\text{expr}}^m C'[(e_0 \ e_1)], \hat{\rho}' \quad q \uparrow_{\text{reach}}^m \text{call}(C[\lambda x.[e]], \hat{c}c_0 :: \hat{\rho}) \quad \hat{c}c_1 := \text{succ}_m(C'[(e_0 \ e_1)], \hat{\rho}') \quad \hat{c}c_1 \sqsubset \hat{c}c_0}{\hat{c}c_1 :: \hat{\rho} / \hat{c}c_0 :: \hat{\rho}}
\end{array}$$

Fig. 5: Demand m -CFA Call Discovery

in the sense that $\hat{c}c_0$ can be instantiated to form $\hat{c}c_1$. Formally, the refinement relation $\sqsubset$ is defined as the least relation satisfying

$$C'[(e_0 \ e_1)] :: \hat{c}c \sqsubset ?_{C[e]} \quad C[(e_0 \ e_1)] :: \hat{c}c_1 \sqsubset C[(e_0 \ e_1)] :: \hat{c}c_0 \iff \hat{c}c_1 \sqsubset \hat{c}c_0$$

If $\hat{c}c_1$ refines $\hat{c}c_0$, *Unknown-Call* does not conclude a $\Downarrow_{\text{call}}^m$ judgment, but rather an *instantiation* judgment $\hat{c}c_1 :: \hat{\rho} / \hat{c}c_0 :: \hat{\rho}$ which denotes that *any* environment $\hat{c}c_0 :: \hat{\rho}$ may be instantiated to $\hat{c}c_1 :: \hat{\rho}$. It is by this instantiation that *Known-Call* will be triggered. When $\hat{c}c_1$ does not refine $\hat{c}c_0$, the resultant caller is ignored which, in effect, filters the callers to only those which are compatible and ensures that Demand m -CFA is indeed context-sensitive.

The $\Downarrow_{\text{call}}^m$ relation relies on a reachability relation $\uparrow_{\text{reach}}^m$ which establishes which queries are (transitively) issued in the course of resolving a top-level query. This relation allows $\Downarrow_{\text{call}}^m$ to instantiate only seen queries with refinement results. This relation is defined over queries themselves; the judgment $q \uparrow_{\text{reach}}^m q'$ captures that, if query q is reachable in analysis, then q' is also, where q is of the form

$$\text{Query} \ni q ::= \text{eval}(C[e], \hat{\rho}) \mid \text{expr}(C[e], \hat{\rho}) \mid \text{call}(C[e], \hat{\rho})$$

Figure 6 presents a formal definition of $\uparrow_{\text{reach}}^m$. The *Reflexivity* rule ensures that the top-level query is considered reachable. The *Ref-Caller* and *Ref-Argument* rules establish reachability corresponding to the *Ref* rule of $\Downarrow_{\text{eval}}^m$: *Ref-Caller* makes the caller query reachable and, if it succeeds, *Ref-Argument* makes the ensuing evaluation query reachable. *App-Operator* and *App-Body* do the same for the *App* rule of $\Downarrow_{\text{eval}}^m$, making, respectively, the operator evaluation query reachable and, if it yields a value, the body evaluation query reachable. *Operand-Operator* makes the evaluation query of the *Operand* rule reachable and *Operand-Body* makes the trace query of any references in the operator body reachable. *Body-Caller-Find* makes the caller query of *Body* reachable; if a caller is found, *Body-Caller-Trace* makes the trace query of that caller reachable. Finally, *Call-Trace* makes sure that the trace query of an enclosing λ of a caller query is reachable.

Figure 7 presents an extension of $\uparrow_{\text{reach}}^m$ which propagates instantiations to evaluation and trace queries. The *Instantiate-Reachable-** rules ensure that if a

$$\begin{array}{c}
\text{REFLEXIVITY} \\
\frac{}{q \uparrow_{\text{reach}}^m q}
\\[10pt]
\text{REF-CALLER} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[x], \hat{\rho}) \quad (C'[e], \hat{\rho}') = \text{bind}(x, C[x], \hat{\rho})}{q \uparrow_{\text{reach}}^m \text{call}(C'[e], \hat{\rho}')}
\\[10pt]
\text{REF-ARGUMENT} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[x], \hat{\rho}) \quad (C'[e], \hat{\rho}') = \text{bind}(x, C[x], \hat{\rho}) \quad C'[e], \hat{\rho}' \Downarrow_{\text{call}}^m C''[(e_0 \ e_1)], \hat{\rho}''}{q \uparrow_{\text{reach}}^m \text{eval}(C''[(e_0 \ e_1)], \hat{\rho}'')}
\\[10pt]
\text{APP-OPERATOR} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[(e_0 \ e_1)], \hat{\rho})}{q \uparrow_{\text{reach}}^m \text{eval}(C[(e_0] \ e_1], \hat{\rho})}
\\[10pt]
\text{APP-BODY} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[(e_0 \ e_1)], \hat{\rho}) \quad C[(e_0] \ e_1], \hat{\rho} \Downarrow_{\text{eval}}^m C'[\lambda x. e], \hat{\rho}'}{q \uparrow_{\text{reach}}^m \text{eval}(C'[\lambda x. e], \text{succ}_m(C[(e_0 \ e_1)], \hat{\rho}) :: \hat{\rho}')}
\\[10pt]
\text{CALL-TRACE} \qquad \qquad \qquad \text{OPERAND-OPERATOR} \\
\frac{q \uparrow_{\text{reach}}^m \text{call}(C[\lambda x. [e]], \hat{c}c :: \hat{\rho})}{q \uparrow_{\text{reach}}^m \text{expr}(C[\lambda x. e], \hat{\rho})} \qquad \frac{q \uparrow_{\text{reach}}^m \text{expr}(C[(e_0 \ e_1)], \hat{\rho})}{q \uparrow_{\text{reach}}^m \text{eval}(C[(e_0] \ e_1], \hat{\rho})}
\\[10pt]
\text{OPERAND-BODY} \\
\frac{q \uparrow_{\text{reach}}^m \text{expr}(C[(e_0 \ e_1)], \hat{\rho}) \quad C[(e_0] \ e_1], \hat{\rho} \Downarrow_{\text{eval}}^m C'[\lambda x. e], \hat{\rho}' \quad x, C'[\lambda x. [e]], \text{succ}_m(C[(e_0 \ e_1)], \hat{\rho}) :: \hat{\rho}' \Rightarrow_{\text{find}}^m C_x[x], \hat{\rho}_x}{q \uparrow_{\text{reach}}^m \text{expr}(C_x[x], \hat{\rho}_x)}
\\[10pt]
\text{BODY-CALLER-FIND} \\
\frac{q \uparrow_{\text{reach}}^m \text{expr}(C[\lambda x. [e]], \hat{\rho})}{q \uparrow_{\text{reach}}^m \text{call}(C[\lambda x. e], \hat{\rho})}
\\[10pt]
\text{BODY-CALLER-TRACE} \\
\frac{q \uparrow_{\text{reach}}^m \text{expr}(C[\lambda x. [e]], \hat{\rho}) \quad C[\lambda x. [e]], \hat{\rho} \Downarrow_{\text{call}}^m C'[(e_0 \ e_1)], \hat{\rho}'}{q \uparrow_{\text{reach}}^m \text{expr}(C'[(e_0 \ e_1)], \hat{\rho}')}
\end{array}$$

Fig. 6: Demand m -CFA Reachability

$$\begin{array}{c}
\text{INSTANTIATE-REACHABLE-EVAL} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[e], \hat{\rho}) \quad \hat{\rho}_1 / \hat{\rho}_0}{q \uparrow_{\text{reach}}^m \text{eval}(C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0])}
\end{array}
\quad
\begin{array}{c}
\text{INSTANTIATE-EVAL} \\
\frac{\hat{\rho}_1 / \hat{\rho}_0 \quad C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0] \Downarrow_{\text{eval}}^m C_v[\lambda x.e], \hat{\rho}_v}{C[e], \hat{\rho} \Downarrow_{\text{eval}}^m C_v[\lambda x.e], \hat{\rho}_v}
\end{array}$$

$$\begin{array}{c}
\text{INSTANTIATE-REACHABLE-EXPR} \\
\frac{q \uparrow_{\text{reach}}^m \text{expr}(C[e], \hat{\rho}) \quad \hat{\rho}_1 / \hat{\rho}_0}{q \uparrow_{\text{reach}}^m \text{expr}(C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0])}
\end{array}
\quad
\begin{array}{c}
\text{INSTANTIATE-EXPR} \\
\frac{\hat{\rho}_1 / \hat{\rho}_0 \quad C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0] \Rightarrow_{\text{expr}}^m C'[(e_0 e_1)], \hat{\rho}'}{C[e], \hat{\rho} \Rightarrow_{\text{expr}}^m C'[(e_0 e_1)], \hat{\rho}'}
\end{array}$$

$$\begin{array}{c}
\text{INSTANTIATE-REACHABLE-CALL} \\
\frac{q \uparrow_{\text{reach}}^m \text{call}(C[e], \hat{\rho}) \quad \hat{\rho}_1 / \hat{\rho}_0}{q \uparrow_{\text{reach}}^m \text{call}(C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0])}
\end{array}
\quad
\begin{array}{c}
\text{INSTANTIATE-CALL} \\
\frac{\hat{\rho}_1 / \hat{\rho}_0 \quad C[e], \hat{\rho}[\hat{\rho}_1 / \hat{\rho}_0] \Downarrow_{\text{call}}^m C'[(e_0 e_1)], \hat{\rho}'}{C[e], \hat{\rho} \Downarrow_{\text{call}}^m C'[(e_0 e_1)], \hat{\rho}'}
\end{array}$$

$$\begin{array}{c}
\text{APP-BODY-INSTANTIATION} \\
\frac{q \uparrow_{\text{reach}}^m \text{eval}(C[(e_0 e_1)], \hat{\rho}) \quad C[(e_0 e_1)], \hat{\rho} \Downarrow_{\text{eval}}^m C'[\lambda x.e], \hat{\rho}'}{\text{succ}_m(C[(e_0 e_1)], \hat{\rho}) :: \hat{\rho}' / ?_x :: \hat{\rho}'}
\end{array}$$

$$\begin{array}{c}
\text{OPERAND-BODY-INSTANTIATION} \\
\frac{q \uparrow_{\text{reach}}^m \text{expr}(C[(e_0 [e_1])], \hat{\rho}) \quad C[(e_0 [e_1])], \hat{\rho} \Downarrow_{\text{eval}}^m C'[\lambda x.e], \hat{\rho}'}{\text{succ}_m(C[(e_0 e_1)], \hat{\rho}) :: \hat{\rho}' / ?_x :: \hat{\rho}'}
\end{array}$$

Fig. 7: Demand m -CFA Instantiation

query of any kind is reachable, then its instantiation is too. When an instantiation $\hat{\rho}_1 / \hat{\rho}_0$ does not apply (so that $\hat{\rho}$ is unchanged), each rule reduces to a trivial inference. The counterpart *Instantiate-** rules, also present in Figure 7, each extend one of $\Downarrow_{\text{eval}}^m$, $\Rightarrow_{\text{expr}}^m$, and $\Downarrow_{\text{call}}^m$ so that, if an instantiated query of that type yields a result, the original, uninstantiated query yields that same result. As discussed at the beginning of this section, Demand m -CFA also discovers instantiations when it extends the environment in the *App* and *Operand* rules. The *App-Body-Instantiation* and *Operand-Body-Instantiation* rules capture these cases.

The definition of Demand m -CFA in terms of an “evaluation” relation (which includes evaluation, trace, and caller resolution) and a reachability relation follows the full formal approach of *Abstracting Definitional Interpreters* by Darais [3]. From this correspondence, we can define the Demand m -CFA resolution of a given query as the least fixed point of these relations, effectively computable with the algorithm Darais [3] provides. We discuss this implementation in more depth in §8.

7 Demand m -CFA Correctness

Demand m -CFA is a hierarchy of demand CFA. Instances higher in the hierarchy naturally have larger state spaces. The size $|\Downarrow_{\text{eval}}^m|$ of the $\Downarrow_{\text{eval}}^m$ relation satisfies

the inequality

$$|\Downarrow_{eval}^m| \leq |Config \times Config| = |Config|^2 = |Exp \times Env|^2 = |Exp|^2 |Env|^2 = n^2 |Env|^2$$

where n is the size of the program. We then have

$$|Env| \leq |Ctx|^n \leq (|Call| + 1)^{mn} \leq n^{mn}$$

since the size of environments is statically bound and may be indeterminate. Thus, $|\Downarrow_{eval}^m| \leq n^{mn+2}$; the state space of $\Downarrow_{eval}^m$ is finite but with an exponential bound; the state spaces of $\Rightarrow_{expr}^m$ and $\Downarrow_{call}^m$ behave similarly.

7.1 Demand m -CFA Refinement

Instances higher in the hierarchy are also more precise, which we formally express with the following theorems.

Theorem 1 (Evaluation Refinement). *If $C[e], \hat{\rho}_0 \Downarrow_{eval}^{m+1} C_v[\lambda x.e_v], \hat{\rho}'_0$ where $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ then $C[e], \hat{\rho}_1 \Downarrow_{eval}^m C_v[\lambda x.e_v], \hat{\rho}'_1$ where $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.*

Theorem 2 (Trace Refinement). *If $C[e], \hat{\rho}_0 \Rightarrow_{expr}^{m+1} C'[(e_0 \ e_1)], \hat{\rho}'_0$ where $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ then $C[e], \hat{\rho}_1 \Rightarrow_{expr}^m C'[(e_0 \ e_1)], \hat{\rho}'_1$ where $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.*

Theorem 3 (Caller Refinement). *If $C[e], \hat{\rho}_0 \Downarrow_{call}^{m+1} C'[(e_0 \ e_1)], \hat{\rho}'_0$ where $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ then $C[e], \hat{\rho}_1 \Downarrow_{call}^m C'[(e_0 \ e_1)], \hat{\rho}'_1$ where $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.*

These theorems state that refining configurations submitted to Demand m -CFA and its successor Demand $m + 1$ -CFA yield refining configurations. The proof proceeds directly (if laboriously) by induction on the derivations of the relations.

7.2 Demand ∞ -CFA and Demand Evaluation

To show that Demand m -CFA is sound with respect to a standard call-by-value (CBV) semantics, we consider the limit of the hierarchy, Demand ∞ -CFA, in which context lengths are unbounded. From here, we bridge Demand ∞ -CFA to a CBV semantics with a concrete form of demand analysis called *Demand Evaluation*. Our strategy is to show that the Demand ∞ -CFA semantics is equivalent to Demand Evaluation which itself is sound with respect to a standard CBV semantics. Demand Evaluation environments, rather than being a sequence of contexts, are sequences of addresses. Like contexts, an address may denote an indeterminate context (i.e. call) which manifests as an address which is not mapped in the store. A store is a pair consisting of a map from addresses to calls and the next address to use; the initial store is $(\perp, 0)$.

Now it is straightforward to express the equivalence between the Demand ∞ -CFA relations and Demand Evaluation.

Theorem 4 (Evaluation Equivalence). *If $\hat{\rho}_0 \hat{\rho} \Leftrightarrow_{\rho} \rho_0, \sigma_0$ then $C[e], \hat{\rho}_0 \Downarrow_{eval}^{\infty} C'[\lambda x.e], \hat{\rho}_1$ iff $C[e], \rho_0, \sigma_0 \Downarrow_{eval}^{de} C'[\lambda x.e], \rho_1, \sigma_1$ where $\hat{\rho}_1 \hat{\rho} \Leftrightarrow_{\rho} \rho_1, \sigma_1$.*

Theorem 5 (Trace Equivalence). *If $\hat{\rho}_0 \hat{\rho} \Leftrightarrow_{\rho} \rho_0, \sigma_0$ then $C[e], \hat{\rho}_0 \Rightarrow_{\text{expr}}^{\infty} C'[(e_0 \ e_1)], \hat{\rho}_1$ iff $C[e], \rho_0, \sigma_0 \Rightarrow_{\text{expr}}^{de} C'[(e_0 \ e_1)], \rho_1, \sigma_1$ where $\hat{\rho}_1 \hat{\rho} \Leftrightarrow_{\rho} \rho_1, \sigma_1$.*

Theorem 6 (Caller Equivalence). *If $\hat{\rho}_0 \hat{\rho} \Leftrightarrow_{\rho} \rho_0, \sigma_0$ then $C[e], \hat{\rho}_0 \Downarrow_{\text{call}}^{\infty} C'[(e_0 \ e_1)], \hat{\rho}_1$ iff $C[e], \rho_0, \sigma_0 \Downarrow_{\text{call}}^{de} C'[(e_0 \ e_1)], \rho_1, \sigma_1$ where $\hat{\rho}_1 \hat{\rho} \Leftrightarrow_{\rho} \rho_1, \sigma_1$.*

These theorems are proved by induction on the derivations, corresponding instantiation of environments on the Demand ∞ -CFA side with mapping an address on the Demand Evaluation side. The technical report [30] presents Demand ∞ -CFA and demand evaluation formally.

8 Evaluation

We implemented Demand m -CFA for a subset of R6RS Scheme [24] including `let`, `let*`, `letrec` binding forms, mutually-recursive definitions and a few dozen primitives. We also implemented support for constructors, numbers, symbols, strings, and characters. We call this language Pure Scheme. Although our analysis does not handle imperative constructs, the analysis can still be deployed in programs with imperative features. It produces sound results for flows that do not experience imperative update and soundly detect when flows do.

We used the *Abstracting Definitional Interpreters* approach [2](ADI) to implement m -CFA and Demand m -CFA analyses for Pure Scheme.

The results of the analysis are obtained through ADI's memoized fixpoint cache. This cache gives control flow results for each segment of control flow independently keyed by the query or subquery. Additionally the cache also has a key for each indeterminate environment which maps to each discovered environment instantiation reachable from the top level query.

We evaluate Demand m -CFA with respect to the following questions:

1. How many queries can Demand m -CFA resolve at a given level of precision (m) and effort allocated per query?
2. How many of those queries yield answers precise enough to support inlining?
3. How does the implementation cost compare to a typical CFA?

To answer the first two questions, we evaluate Demand m -CFA on a set of R6RS programs used by Johnson et al. [9], which is standard within the literature.

8.1 Query Resolution Rate

Like an exhaustive analysis, a demand analysis is subject to client-imposed resource constraints. However, unlike with exhaustive analysis, failure in a demand analysis is not catastrophic: because analysis information is sought selectively and independently through queries, the client can increase the budget and reissue the query or proceed without control flow information.

We evaluate the proportion of queries that Demand m -CFA is able to resolve within specific effort limits and at different context sensitivity levels. Effort is measured in gas, where each subquery consumes one unit.

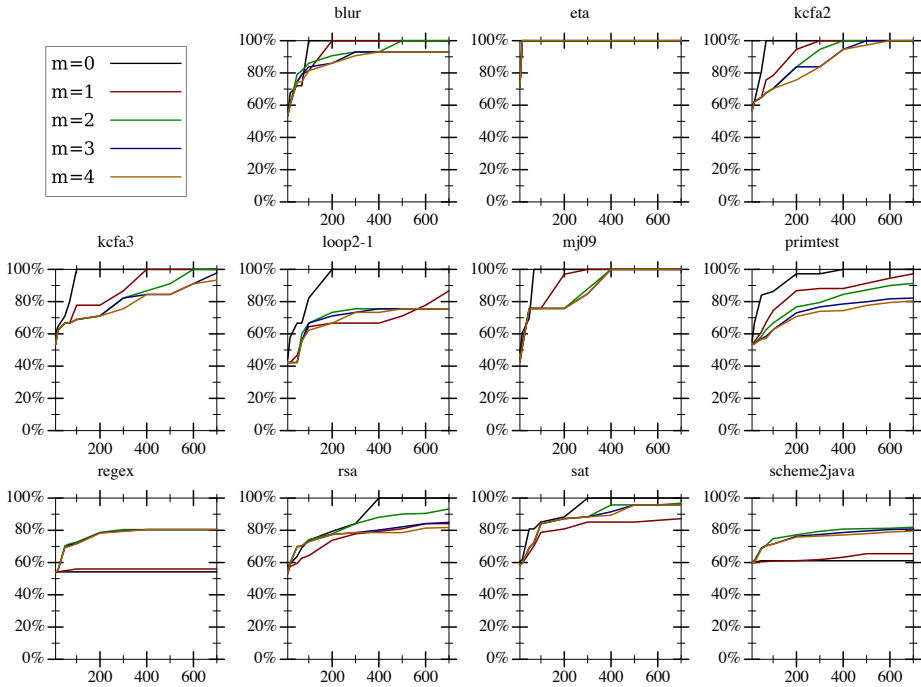


Fig. 8: The percent of queries answered (y-axis) given the per-query gas allocated to Demand m -CFA (x-axis). Many trivial queries (40%-60%) are answered given very little effort, as seen by the immediate jump. Some additional queries are answered with more effort, but some non-trivial queries are answered with less effort by increasing the context sensitivity parameter m (as seen in the larger `regex` and `scheme2java` example programs). Increasing context sensitivity can sometime help the analysis distinguish context-irrelevant flows and avoid spending gas on unrelated parts of the program.

Figure 8 displays the percent of queries answered depending on the effort allotted. The graphs trend upward and to the right, regardless of our choice of m . Notable exceptions include `regex`, and `scheme2java` which both contain highly connected control flow graphs. Both significantly improve when explored at higher m which exemplifies the known paradox of precision from exhaustive CFA, in which increasing precision can sometimes decrease the state space due to fewer spurious flows being merged. Our results show that this phenomenon holds even for subportions of programs. We hypothesize that this paradox might be more useful in the demand context because Demand m -CFA (1) does not attempt to explore undemanded (and therefore query-irrelevant) parts of the program at high levels of context-sensitivity and cost, and (2) only explores the environment instantiations that are required to answer the query, leaving as much of the context indeterminate as possible.

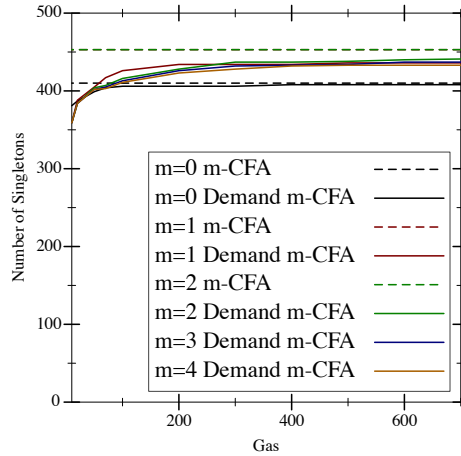


Fig. 9: The number of singleton flow sets (y-axis) found by a Demand m -CFA analysis given the gas allocated per query (x-axis) aggregated across programs. Dashed lines represent the baseline number of singleton flow sets found by an exhaustive exponential m -CFA analysis (graphed as a horizontal line). Dashed lines above $m = 2$ are not shown because `scheme2java` timed out for $m \geq 2$ exhaustive m -CFA. The dashed line for $m = 1$ is underneath $m = 2$.

8.2 Actionable Query Resolution Rate

For many clients, query results must meet a certain level of precision to be useful. For instance, compilers require results to be singleton flow sets in order to inline or otherwise optimize programs. Figure 9 shows the number of singleton flow sets Demand m -CFA was able to obtain for given levels of precision (m) and effort (gas) across the entire program corpus. The total number of singleton flow sets obtained by the corresponding exhaustive m -CFA for each level of precision is also graphed. (Because one program exceeded a 24-hour timeout we do not include results of exhaustive m -CFA for $m > 2$. A breakdown of results per program is given in the technical report [30].)

The data show that Demand m -CFA generally finds most of the singleton value flows that exhaustive m -CFA does, and support that most actionable control flow information is available at low effort bounds. The data also indicate that the attained level of context sensitivity is mostly independent of the effort allotted. This is consistent with Demand m -CFA's ability to avoid instantiating indeterminate aspects of environments when they are not relevant. In contrast, exhaustive m -CFA fully instantiates all contexts which often leads to an explosion in the explored state space.

As Figure 9 shows, Demand m -CFA is not able to discover all the singleton flow sets as the exhaustive m -CFA at any level of effort bound tested. Because of its reachability assumptions, Demand m -CFA finds more flows than exhaustive

m-CFA, which reduces the number of singleton flows it detects. For instance, if Demand *m*-CFA is finding the callers of `f` in the expression `(λ (g) (f 42))` to discover arguments to `f`, it assumes that `(f 42)` is reachable—i.e., it assumes that `(λ (g) (f 42))` is called. If that assumption is false, then the argument 42 does not actually contribute to the value that Demand *m*-CFA is resolving, and its inclusion is manifest as imprecision. The data indicate, that in practice, this reduction is less than 5% across the program corpus.

8.3 Implementation Cost

There are two notions of cost that we examine in this section: (1) the raw cost of implementation on a small core language relative to a corresponding exhaustive analysis, and (2) the actual cost of implementation for a real-world language.

To evaluate the first notion of cost, we implemented both exhaustive and Demand *m*-CFA for Pure Scheme. The codebases shared approximately 1000 lines of code. Exhaustive *m*-CFA required approximately 450 lines of additional code. Demand *m*-CFA required approximately 630 lines of additional code. For Pure Scheme the total cost of implementing Demand *m*-CFA is not significantly higher than that of exhaustive *m*-CFA.

To evaluate the second notion of cost, we implemented Demand *m*-CFA for the subset of the Koka language which does not include algebraic effect handlers and mutation. This highlights a strength of Demand *m*-CFA that, despite being implemented for only a subset of Koka, our implementation is still useful. For example, we integrated it into a language server which is able to provide low-latency control flow information to clients. In contrast, to implement an exhaustive analysis for the same purpose, one must implement the full set of Koka’s features. While exhaustive analyses support mutation without issue, support for algebraic effect handlers is not currently tractable. Our implementation, written in Haskell, consists of approximately 3000 lines of code.

Our experience with the Koka compiler indicates that integrating Demand *m*-CFA into a real-world compiler or language server is in some cases *more* tractable than an exhaustive analysis, since not all language features or primitives need to be supported before getting useful and actionable results.⁴

9 Related Work

The original inspiration for demand CFA is demand dataflow analysis [8] which refers to dataflow analysis algorithms which allow selective, local, parsimonious analysis of first-order programs driven by the user. Demand CFA refers to a class of algorithms with those same characteristics which operate in the presence of first-class functions. This work extends Demand OCFA [5], currently the sole embodiment of demand CFA, with context sensitivity using a context abstraction similar to that of *m*-CFA [15].

⁴ The Pure Scheme and Koka implementations are available on GitHub[29,28].

The most-closely related work is that of Schoepe et al. [20] which utilizes first-order forward and backward analyses to construct call graphs on demand to compose a higher-order analysis. Although it treats control with context sensitivity, it does not model or track the environment structure of higher-order objects. Our work has a similar goal—to achieve demand-driven analysis of higher-order programs—but it has sought to provide a concise analysis which models both the control and environment structure of evaluation with equal facility.

DDPA [17] is nominally a context-sensitive, demand-driven analysis for higher-order programs. However, before resolving any on-demand queries, DDPA must bootstrap a global control-flow graph to support them. Because of this large, fixed, up-front cost, DDPA does not provide the pricing model expected of a demand analysis.

Several other “demand-driven” analyses exist for functional programs. Midtgaard and Jensen [13] present a “demand-driven 0-CFA” derived via a calculational approach which analyzes only those parts of the program on which the *program’s* result depends. Biswas [1] presents a demand analysis that takes a similar approach to “demand-driven 0-CFA” to analyze *first-order* functional programs. These analyses are certainly “demand-driven” in the sense the authors intend (demanded by the full program), but not in the sense that we use it, as a descriptor for generic demand CFA. Our Demand CFA can be considered a loose extension in spirit which analyzes only those parts of a *higher-order* functional program on which a *selected expression’s* result depends.

Nicolay et al. [16] produce a flow analysis where new information is propagated between interprocedural and intraprocedural parts of the program via the store. As such they incrementally build a reactive control flow graph on demand. In addition to read and write dependency tracking their analysis explicitly tracks call effects which triggers a demand for functions stored at that point in the store to be evaluated interprocedurally with new arguments. However, while they present the theory for a context sensitive analysis, they do not present results for context sensitivity, and while their approach could be adapted to a demand based setting, their focus is on full program analysis.

Heintze and McAllester [6] describe the “demand-driven” subtransitive CFA which computes an underapproximation of control flow in linear time and can be transitively closed for an additional quadratic cost. Their analysis exploits type structure and applies to typed programs with bounded type, whereas our formulation neither considers nor requires types.

Points-to analysis is the analogue of CFA in an object-oriented setting in the sense that both are fundamental analyses that provide the necessary support for higher-level analysis. Many context-sensitive demand-driven points-to analyses (e.g. Lu et al. [12] Shang et al. [21] Späth et al. [23] Su et al. [26]) exist, formulated for Java. Though both points-to analysis and CFA target higher-order programs, Might et al. [15] observed that the explicit object creation in the object-oriented setting induces flat closures whereas implicit closure creation in the functional setting induces nested closures. Moreover, mutation is routine in many object-oriented settings (e.g. Java) in which points-to analyses are ex-

pressed. Both nested environments and the prevalence of immutable variables are fundamental to our realization of demand CFA.

Pointer analysis is the imperative analogue of control-flow analysis and it too enjoys a rich literature. When function pointers are present, a demand-driven pointer analysis must be able to reconstruct the call graph on the fly, a requirement shared by demand-driven CFA.

Heintze and Tardieu [7] present a demand-driven pointer analysis for C capable of constructing the call graph on the fly. They recognize that most call targets are not specified indirectly through pointers and advocate demand-driven analysis to resolve indirect targets when they appear. In the core language of Demand *m*-CFA, calls with indirect targets are commonplace; the language was chosen to emphasize those cases in particular. In a full-featured language, calls with indirect targets—though still more common than in C—are less common, since calls to primitives are typically direct and statically discernible. However, closures are not merely code pointers, but environments too, and we have had to exercise great care to ensure that Demand *m*-CFA correctly models their behavior.

Saha and Ramakrishnan [19] formulate points-to analysis of C as a logic program, so that the incremental-evaluation capabilities of the logic engine yield an incremental analysis. They combine an incremental and demand-driven approach to update the points-to model in response to changes in the program. Our work is similar in that it may be possible to cast it as a logic program and thereby combine an initial exhaustive CFA with an incremental, demand-driven CFA to keep the model synchronized with a changing program.

More recently, Sui and Xue [27] developed SUPA, an on-demand analyzer for C programs which refines value flows during analysis. SUPA is designed for low-budget environments and its evaluation explicitly weighs its ultimate precision against its initial budget. Demand *m*-CFA can be positioned similarly where the low-budget environments are compilers and IDEs for functional programs.

10 Future Work

Having established context sensitivity for Demand CFA, we intend to: (1) develop a general operational framework in which a variety of effects can be expressed, including mutation, (2) investigate the tradeoff between precision and the non-detection of dead code within the analysis (occurring during binding resolution), (3) determine whether *m*-CFA's re-binding policy could be adopted to a demand setting, and whether Demand *m*-CFA could benefit from it, (4) determine how to reuse fixpoint caches between queries while keeping environments as indeterminate as possible to create an incremental analysis [25,18], (5) consider how selective context sensitivity [11] could be realized given the indeterminate environments of our approach.

References

1. Biswas, S.K.: A demand-driven set-based analysis. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 372–385. POPL '97, ACM, New York, NY, USA (Jan 1997). <https://doi.org/10.1145/263699.263753>
2. Darais, D., Labich, N., Nguyen, P.C., Van Horn, D.: Abstracting definitional interpreters (functional pearl). Proceedings of the ACM on Programming Languages **1**(ICFP), 12:1–12:25 (Aug 2017). <https://doi.org/10.1145/3110256>
3. Darais, D.C.: Mechanizing Abstract Interpretation. Ph.D. thesis, University of Maryland, College Park, MD, USA (2017)
4. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **19**(6), 992–1030 (Nov 1997). <https://doi.org/10.1145/267959.269970>
5. Germane, K., McCarthy, J., Adams, M.D., Might, M.: Demand control-flow analysis. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 11388, pp. 226–246. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_11
6. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation. pp. 261–272. PLDI '97, ACM, New York, NY, USA (May 1997). <https://doi.org/10.1145/258915.258939>
7. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 24–34. PLDI '01, ACM, New York, NY, USA (Jun 2001). <https://doi.org/10.1145/378795.378802>
8. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. ACM SIGSOFT Software Engineering Notes **20**(4), 104–115 (1995)
9. Johnson, J.I., Sergey, I., Earl, C., Might, M., Van Horn, D.: Pushdown flow analysis with abstract garbage collection. Journal of Functional Programming **24**(2–3), 218–283 (2014). <https://doi.org/10.1017/S0956796814000100>
10. Leijen, D.: Koka programming language. <https://github.com/koka-lang/koka> (2019), accessed: 2024-05-30
11. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: A principled approach to selective context sensitivity for pointer analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**(2), 1–40 (2020)
12. Lu, Y., Shang, L., Xie, X., Xue, J.: An incremental points-to analysis with CFL-reachability. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction. Lecture Notes in Computer Science, vol. 7791, pp. 61–81. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37051-9_4
13. Midtgaard, J., Jensen, T.: A calculational approach to control-flow analysis by abstract interpretation. In: Alpuente, M., Vidal, G. (eds.) Static Analysis. Lecture Notes in Computer Science, vol. 5079, pp. 347–362. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_23
14. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: abstract garbage collection and counting. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming. pp. 13–25. ICFP '06, ACM, New York, NY, USA (Sep 2006). <https://doi.org/10.1145/1159803.1159807>

15. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k -CFA paradox: illuminating functional vs. object-oriented program analysis. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 305–315. PLDI '10, ACM, New York, NY, USA (Jun 2010). <https://doi.org/10.1145/1806596.1806631>
16. Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C.: Effect-driven flow analysis. In: *Verification, Model Checking, and Abstract Interpretation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings 20*. pp. 247–274. Springer (2019)
17. Palmer, Z., Smith, S.F.: Higher-order demand-driven program analysis. In: Krishnamurthi, S., Lerner, B.S. (eds.) *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 56, pp. 19:1–19:25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). <https://doi.org/10.4230/LIPIcs.ECOOP.2016.19>
18. Van der Plas, J., Stiévenart, Q., Van Es, N., De Roover, C.: Incremental flow analysis through computational dependency reification. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. pp. 25–36 (2020). <https://doi.org/10.1109/SCAM51674.2020.00008>
19. Saha, D., Ramakrishnan, C.R.: Incremental and demand-driven points-to analysis using logic programming. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. pp. 117–128. PPDP '05, ACM, New York, NY, USA (Jul 2005). <https://doi.org/10.1145/1069774.1069785>
20. Schoepe, D., Seekatz, D., Stoilkovska, I., Stucki, S., Tattersall, D., Bolignano, P., Raimondi, F., Chang, B.Y.E.: Lifting on-demand analysis to higher-order languages. In: *International Static Analysis Symposium*. pp. 460–484. Springer (2023)
21. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. pp. 264–274. CGO '12, ACM, New York, NY, USA (Mar 2012). <https://doi.org/10.1145/2259016.2259050>
22. Shivers, O.: *Control-Flow Analysis of Higher-Order Languages*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
23. Späth, J., Do, L.N.Q., Ali, K., Bodden, E.: Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In: Krishnamurthi, S., Lerner, B.S. (eds.) *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18–22, 2016, Rome, Italy*. *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 56, pp. 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). <https://doi.org/10.4230/LIPIcs.ECOOP.2016.22>
24. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A., Findler, R., Matthews, J.: Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press, New York, NY, USA, 1st edn. (Jun 2010)
25. Stein, B., Chang, B.Y.E., Sridharan, M.: Demanded abstract interpretation. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. pp. 282–295 (2021)
26. Su, Y., Ye, D., Xue, J.: Parallel pointer analysis with CFL-reachability. In: *43rd International Conference on Parallel Processing (ICPP 2014)*. pp. 451–460. IEEE Computer Society, Los Alamitos, CA, USA (2014). <https://doi.org/10.1109/ICPP.2014.54>

27. Sui, Y., Xue, J.: On-demand strong update analysis via value-flow refinement. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 460–473. FSE 2016, ACM, New York, NY, USA (Nov 2016). <https://doi.org/10.1145/2950290.2950296>
28. Whiting, T.: Demand m-cfa koka implementation. <https://github.com/TimWhiting/koka/tree/esop-2025> (2024)
29. Whiting, T.: Demand m-cfa pure scheme implementation. <https://github.com/TimWhiting/context-sensitive-demand-cfa> (2024)
30. Whiting, T., Germane, K.: Demand m-cfa technical report. <https://timwhiting.dev/assets/pdfs/demand-mcfa-techreport.pdf> (2024)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





On the Relationship between Dijkstra Monads and Higher-Order Fixpoint Logic

Risa Yamada¹, Naoki Kobayashi¹, Ken Sakayori¹, and Ryosuke Sato²

¹ The University of Tokyo, Tokyo, Japan

{r.yamada,koba,sakayori}@is.s.u-tokyo.ac.jp

² Tokyo University of Agriculture and Technology, Tokyo, Japan

rsato@acm.org

Abstract. We study the relationship between two approaches to higher-order program verification: a semi-automated method using Dijkstra monads and a fully automated method using a higher-order fixpoint logic called HFL(Z). Although the origins of both approaches are quite different, there are some striking similarities: both convert programs to corresponding predicate transformers, and the conversion is essentially obtained by a CPS transformation. After reviewing the two approaches, we formalize an exact correspondence between the two for a restricted fragment of a functional language. We also point out that, outside the restricted fragment, there are some important differences between the two approaches, suggesting the need for cross-fertilization to obtain the best of the two approaches. As an example of the cross-fertilization, we also propose a semi-automated verification method, which requires less annotations than the Dijkstra monad approach and can scale to larger programs than the HFL(Z) approach.

1 Introduction

This paper studies the relationship between two approaches to the verification of higher-order functional programs: one based on Dijkstra monads [15,1,12] and the other based on HFL(Z) [9,17].

The Dijkstra monad approach, introduced by Swamy et al. [15], is a semi-automated verification method for programs with higher-order functions and monadic effects. It associates a program with a corresponding predicate transformer in the form of a monad. Using the monad, the verification condition for a program can be computed through type inference, and discharged by SMT solvers. The method is only *semi*-automated; the type inference procedure requires type annotations for recursive functions. Kobayashi et al.'s approach based on HFL(Z) [9,17] is a fully-automated approach to the verification of temporal properties of higher-order programs. In this approach, a functional program is automatically transformed to a formula of HFL(Z), a higher-order logic with fixpoint operators and integer arithmetic, so that the program satisfies a given temporal property just if the formula is valid. The validity of the formula is then checked by an automated solver for HFL(Z) validity checking [6,5,8].

The two approaches above are seemingly different, but there are some strong similarities. Consider the successor function defined by `let succ x = x+1`.³ The Dijkstra monad approach (DM approach, for short) assigns the type $x : \text{int} \rightarrow \mathsf{D} \text{ int } \lambda p.p(x+1)$ to the function, where $\mathsf{D} \text{ int } \lambda p.p(x+1)$ is a monadic type indexed by the predicate transformer $\lambda p.p(x+1) : (\text{int} \rightarrow \text{prop}) \rightarrow \text{prop}$. The type means that the function takes an integer x , and for the return value r to satisfy the postcondition $p(r)$, it suffices that the precondition $p(x+1)$ holds. Thus, if we wish to prove $\forall x.x > 0 \Rightarrow \text{succ } x > 0$, it suffices to prove $\forall x.x > 0 \Rightarrow (\lambda p.p(x+1))(\lambda r.r > 0)$, which is equivalent to $\forall x.x > 0 \Rightarrow x+1 > 0$.⁴

In the HFL(Z) approach, for the same program, the call-by-value CPS transformation is first applied to obtain a corresponding call-by-name program `let succ_cps x k = k(x+1)`. Then the function `succ_cps` is converted to the corresponding predicate $\text{Succ} := \lambda x.\lambda k.k(x+1)$, and the specification $\forall x.x > 0 \Rightarrow \text{succ } x > 0$ is mapped to the HFL(Z) formula $\forall x.x > 0 \Rightarrow \text{Succ } x (\lambda r.r > 0)$. In the example above, one can observe that the predicate $\text{Succ } x$ is essentially a predicate transformer of the original function `succ`; indeed, $\text{Succ } x = (\lambda x.\lambda k.k(x+1))x$ is $\alpha\beta$ -equal to the predicate transformer $\lambda p.p(x+1)$ obtained in the DM approach. The key translation used to obtain Succ was a CPS transformation.⁵ Incidentally, Ahman et al. [1] have shown that various Dijkstra monads can automatically be obtained from effect monads by using the CPS transformation. In spite of these similarities, the two approaches have been studied independently.

In this paper, we first formalize a precise correspondence between the DM and HFL(Z) approaches, for a restricted fragment of a higher-order functional language parameterized by effect monads. We show that, for every predicate transformer derived in the HFL(Z) approach, an equivalent predicate transformer can be obtained also in the DM approach, and that, conversely for every predicate transformer derivable in the DM approach, an at least equally good predicate transformer can be obtained in the HFL(Z) approach.⁶ We also show that, outside the restricted fragment, there are at least two important differences between the two approaches. One is for programs containing “nested monads”, for which we point out some awkwardness in the DM approach: information about the program behavior is scattered and replicated in types. The other difference is for recursive programs: in the DM approach, a user has to specify the type of every recursive function, while in the HFL(Z) approach, there is no such need and

³ The Dijkstra monad is used to handle effectful programs, but we consider here a pure functional program for the sake of simplicity; see later sections for more details about the Dijkstra monad.

⁴ In this particular case, we can obtain the same condition by just unfolding the definition of `succ` in $\forall x.x > 0 \Rightarrow \text{succ } x > 0$. It does not work, however, in general, when the function is recursively defined or when its code is unavailable and only the type is known.

⁵ In the case of a stateful program, one should also apply a store-passing transformation first, to obtain a pure functional program. Such examples are given later.

⁶ This slight asymmetry of the correspondences is due to the nondeterminism of type derivations in the DM approach.

predicate transformers are computed fully automatically, thanks to the use of a higher-order fixpoint logic. The latter difference yields some trade-offs: whilst the fully automatic approach of $\text{HFL}(\mathbf{Z})$ frees users from burdens of annotating programs, it imposes a heavy burden on the backend $\text{HFL}(\mathbf{Z})$ solver and thus makes the verification time highly unpredictable. Based on the latter difference, we also propose an integration of the two approaches, to take the best of two worlds. In the integrated approach, a user may provide type annotations for recursive functions *selectively*; thanks to the use of the fixpoint logic, not all recursive functions need to be annotated. Based on a refinement type system for $\text{HFL}(\mathbf{Z})$ [6], we can automatically extract verification conditions from *partially* annotated programs. We have implemented a prototype tool, and confirmed its effectiveness: there are programs for which the verification tool based on the original $\text{HFL}(\mathbf{Z})$ approach times out, but our tool succeeds without (unlike the DM approach) requiring full annotations for recursive functions.

Our main contributions are summarized as follows.

- A formalization of the correspondence between the DM approach and the $\text{HFL}(\mathbf{Z})$ approach, which have been studied independently.
- Analysis of the differences between the two approaches, which suggests cross-fertilization of the two approaches.
- A proposal of a semi-automated verification method to get the best of the two approaches. Our method is more scalable and predictable than the $\text{HFL}(\mathbf{Z})$ approach, and works with fewer annotations than the DM approach.

In addition to these main contributions, our formalization includes generalized translation rules for the $\text{HFL}(\mathbf{Z})$ approach to handle monadic effects, extending the example-based sketch of Kobayashi et al. [9,17].

The rest of this paper is structured as follows. Section 2 reviews the two approaches based on Dijkstra monads and $\text{HFL}(\mathbf{Z})$ and informally observes their similarities. Section 3 considers a restricted fragment of a higher-order functional language, and proves an exact correspondence between the two approaches. Section 4 discusses the differences between the two approaches. Section 5 presents our semi-automated verification method and reports preliminary experimental results. Section 6 discusses related work and Section 7 concludes the paper.

2 Overview of Dijkstra Monads, $\text{HFL}(\mathbf{Z})$, and Their Relationship

This section reviews the Dijkstra monad approach and the $\text{HFL}(\mathbf{Z})$ approach for higher-order program verification, and observes their similarities.

2.1 The Dijkstra Monad Approach

As mentioned already, Swamy et al. [15] introduced a semi-automated verification method targeting higher-order programs with monadic effects, based on a monad of predicate transformers called the *Dijkstra monad*. The types of effectful

computations has the form $D \ t \ w$, where D is the Dijkstra monad defined for each kind of effect, t is the pure type of the result, and w is a predicate transformer that describes the behavior of the computation by mapping postconditions to preconditions.

For example, consider the following program, which manipulates a state of type `int`.

```
let increment () = let x = get () in put (x+1)
```

The function `increment` gets the current state x and writes back $x + 1$, thus incrementing the state. The function is given the type:

$$\text{unit} \rightarrow D^{\text{STATE}} \text{unit} (\lambda s_{\text{init}}, \text{post}. \text{post} () (s_{\text{init}} + 1)).$$

The predicate transformer $\lambda s_{\text{init}}, \text{post}. \text{post} () (s_{\text{init}} + 1)$ has the type $\text{int} \rightarrow (\text{unit} \rightarrow \text{int} \rightarrow \text{prop}) \rightarrow \text{prop}$. The first parameter s_{init} stands for the initial state, and the second parameter post is the postcondition, which is a binary predicate on the result value and the final state. Given the parameters, the predicate transformer returns a precondition that guarantees the postcondition to hold after the computation. For example, to prove that the final state is positive after calling `increment`, it suffices to apply the predicate transformer to the postcondition $\lambda(). \lambda s_{\text{final}}. s_{\text{final}} > 0$, and prove that the computed precondition $s_{\text{init}} + 1 > 0$ holds before the call.

To derive a predicate transformer from a program, Swamy et al. [15] proposed a kind of a dependent type system and an inference algorithm. For programs without recursion, predicate transformers and the associated verification conditions can automatically be inferred, although users may optionally provide type annotations to specify the behavior of sub-expressions. For recursive functions, however, those annotations must be provided, as in other deductive (verification-condition-generation) approaches to program verification.

The type system is parametric in the choice of effects, so that the method can be applied to various effects as long as the definition of the Dijkstra monad is given, which consists of the predicate transformers of the monadic return and bind operators and additional effect actions. For example, to reason about effectful computations with an integer state, users just need to provide the following definitions; the system can then automatically (except for recursive functions) infer predicate transformers.

$$\begin{aligned} D^{\text{STATE}} : t : \text{type} &\rightarrow w : W^{\text{STATE}} \ t \rightarrow \text{type} \\ W^{\text{STATE}} \ t = \text{int} &\rightarrow (t \rightarrow \text{int} \rightarrow \text{prop}) \rightarrow \text{prop} \\ \text{return}^{\text{STATE}} : x : t &\rightarrow D^{\text{STATE}} \ t (\lambda s. \lambda p. \text{p } x \ s) \\ \text{bind}^{\text{STATE}} : m : (D^{\text{STATE}} \ t_1 \ w_1) &\rightarrow f : (x : t_1 \rightarrow D^{\text{STATE}} \ t_2 \ (w_2 \ x)) \rightarrow \\ &D^{\text{STATE}} \ t_1 \ (\lambda s_0. \lambda p. w_1 s_0 (\lambda x. \lambda s_1. w_2 \ x \ s_1 \ p)) \\ \text{get} : \text{unit} &\rightarrow D^{\text{STATE}} \ \text{int} (\lambda s. \lambda p. \text{p } s \ s) \\ \text{put} : x : \text{int} &\rightarrow D^{\text{STATE}} \ \text{unit} (\lambda s. \lambda p. \text{p } () \ x). \end{aligned}$$

Here, the first line is the kind of the Dijkstra monad and the second line defines $\mathbf{W}^{\text{STATE}} t$, the type of predicate transformers. The third and fourth lines define the types of monad operators, which specify how predicate transformers can be composed from those of sub-computations. The last two lines define predicate transformers for primitive operations on states. Actually, these definitions can be obtained partially automatically through the CPS transformation [1,12]. The Dijkstra monad approach has been implemented in F^* , and applied to the verification of real-world programs [2].

2.2 The HFL(Z) Approach

Kobayashi et al. [9,17] proposed an approach to higher-order program verification based on HFL(Z), an extension of Viswanathan and Viswanathan's higher-order fixpoint logic [16] with integers. In that approach, a higher-order functional program with a regular-temporal-property specification is automatically translated into a formula of HFL(Z), so that the program satisfies the property just if the formula is valid. The validity of the formula is undecidable in general, but automated (but inevitably incomplete due to the undecidability) validity checkers for HFL(Z) have been implemented since then [6,5,8]; the approach thus realizes fully automated verification of higher-order programs.

By using fixpoint operators, the properties of recursive functions can be precisely expressed in the HFL(Z) approach, without any need for manual annotations. For example, consider the following function.

```
let rec sum n = if n=0 then 0 else n + sum (n-1)
```

The function `sum` computes the sum of the first n natural numbers if n is a non-negative integer, and diverges otherwise. If we are interested in the total correctness, it suffices to translate the function to the predicate `Sum` defined by the following fixpoint equation:

$$\text{Sum} =_{\mu} \lambda n. \lambda k. (n = 0 \Rightarrow k\ 0) \wedge (n \neq 0 \Rightarrow \text{Sum}\ (n - 1)\ (\lambda s. k\ (n + s))).$$

Here, the equation defines `Sum` as the *least* predicate that satisfies the equality. The above equation is automatically obtained from the function definition by applying the CPS transformation (note that k above represents a continuation), and by encoding the conditional expression into the corresponding logical formula. `Sum` n can be viewed as a predicate transformer; it takes a postcondition k , and returns the corresponding precondition on n for `sum` to terminate and for the return value r to satisfy $k\ r$. The total correctness is thus expressed as $\forall n \geq 0. \text{Sum}\ n\ (\lambda r. r = n(n + 1)/2)$. If we are interested in partial correctness instead, then we just need to replace $=_{\mu}$ with $=_{\nu}$ in the above fixpoint equation, so that `Sum` is defined as the *greatest* predicate that satisfies the equation.

The HFL(Z) approach targets a higher-order functional language with general recursion, nondeterminism, and an abstract notion of events, but various other effects (such as states and exceptions) can be encoded by a combination with standard transformations for functional programs such as store passing

translation and variants of CPS transformations. For example, to deal with the **increment** function in Section 2.1, we first apply the store-passing transformation (and inline **get** and **put**) to obtain:

```
let increment () s = let x = s in (), x+1
```

We can then obtain the following HFL(Z) predicate **Inc** by using the CPS (and currying) transformation:

$$\text{Inc} := \lambda().\lambda s.\lambda k.k () (s + 1).$$

Suppose we wish to prove that, given a non-negative initial state, the state after calling **increment** is positive. That is expressed by the formula $\forall s \geq 0. \text{Inc} () s (\lambda().\lambda s'.s' > 0)$. Here, notice that $\text{Inc} () =_{\beta} \lambda s.\lambda k.k () (s + 1)$ matches the predicate transformer obtained in the DM approach in Section 2.1.

2.3 The Similarity between Dijkstra Monads and HFL(Z)

We have already seen similarities between the two approaches for first-order functions (**succ** in Section 1 and **increment** in this section). We discuss a higher-order case in this subsection. Let us consider the following twice function.

```
let twice f x = let y = f x in f y
let plustwo = twice succ
```

In the DM approach, the function **twice** can be given the following type.

$$\forall w_f.f : (x : \text{int} \rightarrow \mathsf{D} \text{int} (w_f x)) \rightarrow x : \text{int} \rightarrow \mathsf{D} \text{int} (\lambda p.w_f x (\lambda y.w_f y p)).$$

Here, the type is polymorphic on the predicate transformer w_f for f . Using w_f , the precondition for the part $f y$ (with respect to the postcondition p) is expressed as $w_f y p$, and thus the predicate transformer for **twice** can be expressed as $\lambda p.w_f x (\lambda y.w_f y p)$. Since **succ** has type $x : \text{int} \rightarrow \mathsf{D} \text{int} (\lambda p.p(x + 1))$, we obtain the type $x : \text{int} \rightarrow \mathsf{D} \text{int} (\lambda p.w_{\text{succ}} x (\lambda y.w_{\text{succ}} y p))$ for **plustwo**, where $w_{\text{succ}} := \lambda x.\lambda p.p(x + 1)$. By reducing it, we obtain $x : \text{int} \rightarrow \mathsf{D} \text{int} (\lambda p.p(x + 2))$. In the HFL(Z) approach, the function **twice** is converted to the following predicate **Twice**:

$$\text{Twice} := \lambda f.\lambda x.\lambda k.f x (\lambda y.f y k).$$

The part $\lambda k.f x (\lambda y.f y k)$ corresponds to $\lambda p.w_f x (\lambda y.w_f y p)$ from the DM approach. The function **plustwo** is converted to **Twice Succ**, which can be simplified as follows.

$$\text{Twice Succ} = (\lambda f.\lambda x.\lambda k.f x (\lambda y.f y k))(\lambda x.\lambda k.k(x + 1)) =_{\beta} \lambda x.\lambda k.k(x + 2).$$

The part $\lambda k.k(x + 2)$ corresponds to $\lambda p.p(x + 2)$. Thus, there are some correspondences between the DM and HFL(Z) approaches even at higher-order.

Table 1 summarizes the correspondence between the two approaches we have seen so far (with some α -renaming). As clear from the table, HFL(Z) formulas can actually be extracted from types of Dijkstra monads. In a higher-order case, HFL(Z) formulas correspond to polymorphic types on predicate transformers. A formal connection is established in the next section.

Function	DM types	HFL(Z) formulas
succ	$x : \text{int} \rightarrow \text{D int } \lambda p.p(x+1)$	$\lambda x.\lambda p.p(x+1)$
increment	$\text{unit} \rightarrow \text{D}^{\text{STATE}} \text{unit } (\lambda s.\lambda p.p \ () \ (s+1))$	$\lambda () . \lambda s.\lambda p.p \ () \ (s+1)$
twice	$\forall w_f.f : (x : \text{int} \rightarrow \text{D int } w_f x) \rightarrow$ $x : \text{int} \rightarrow \text{D int } \lambda p.w_f x (\lambda y.w_f y p)$	$\lambda f.\lambda x.\lambda p.f x \lambda y.f y p$

Table 1. Correspondence between DM types and HFL(Z) formulas

3 Correspondence between the Two Approaches

In the previous section, we reviewed the two approaches to higher-order program verification and observed their similarities informally. In this section, we formalize a concrete correspondence between the two approaches for a restricted fragment of a higher-order functional language.

We first set up a common language for comparing the two approaches in Section 3.1. We then formalize the derivation of predicate transformers in the DM approach in Section 3.2, and the derivation of logical formulas in the HFL(Z) approach in Section 3.3. We then state and prove a correspondence in Section 3.4.

3.1 Language

This subsection introduces a simple call-by-value, effectful higher-order functional language, which shall be used as the common language for formally comparing the two approaches.

We exclude recursion and restrict the shape of types; as discussed in Section 4, some important differences show up for recursion and certain types.

The syntax of (monadic) types is given as follows:

$$a ::= \text{unit} \mid \text{int} \quad t ::= s \rightarrow r \quad r ::= M a \mid t \quad s ::= a \mid t$$

Here, a is a base type for pure expressions, $M a$ represents the type of effectful computations that return values of type a . Inheriting the restrictions from the two approaches, the syntax of types above is restrictive compared to ordinary monadic type systems. As in the DM approach, a monadic type $M a$ can only appear as the co-domain of a function type. The restriction of the parameter of M to a base type a comes from the restriction on the side of HFL(Z); this point will be discussed in Section 4. In addition to the restrictions originating from the two approaches, our formalization of the correspondence requires that monads must not be nested: for instance, $\text{int} \rightarrow M (\text{int} \rightarrow M \text{int})$ is rejected. These restrictions still allow some higher-order function types: for example, $(\text{int} \rightarrow M \text{int}) \rightarrow \text{int} \rightarrow M \text{int}$ is allowed. Further discussion on the restrictions can be found in Section 4.

The syntax of expressions is given by:

$$e ::= () \mid n \mid x \mid \text{op}(\tilde{e}) \mid \lambda x : s. e \mid e_1 e_2 \mid \text{if } \text{pred}(\tilde{e}) \text{ then } e_1 \text{ else } e_2 \\ \mid \text{return } e \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{call}^{act}$$

Here, n ranges over the set of integers, op ranges over a set of operations on integers (such as addition), $\tilde{e}$ denotes a sequence $e_1, \dots, e_k$ of expressions, and pred ranges over a set of predicates on integers (such as equality, and nullary predicates **true** and **false**). We write $|\tilde{e}|$ for the length of the sequence $\tilde{e}$, and write $\text{arity}(\text{op})$ and $\text{arity}(\text{pred})$ for the arity of op and pred respectively. We just write pred for $\text{pred}()$ when pred is nullary. The set of expressions includes variables, λ -abstractions, applications, conditionals, a monadic return operator, and let-expressions. The expression call^{act} invokes the effect action named act , where act ranges over a set A of names of effect actions. We may omit the type annotations for λ -abstractions when they can be inferred from the context. Recursion is not included here, but will be discussed in Section 4.

We introduce a type system to reject inappropriate expressions such as $1 + ()$. We call this type system the *simple (monadic) type system*, to distinguish it from the type system for Dijkstra monads introduced later. The typing rules are given in Figure 1, which define two kinds of judgments $\Gamma \vdash e : s$ and $\Gamma \vdash e : \mathbb{M} a$, where the latter is for effectful computations. In the rules M-OP and M-IF, $\Gamma \vdash \tilde{e} : \widetilde{\text{int}}$ means $\Gamma \vdash e_i : \text{int}$ for every $i \in \{1, \dots, k\}$. In the rule M-ACT, t^{act} denotes the type of the effect action act (which is introduced below).

$\frac{}{\Gamma \vdash () : \text{unit}}$	(M-UNIT)	$\frac{\Gamma \vdash e_1 : s \rightarrow r \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : r}$	(M-APP)
$\frac{}{\Gamma \vdash n : \text{int}}$	(M-INT)	$\frac{\Gamma \vdash e : a}{\Gamma \vdash \text{return } e : \mathbb{M} a}$	(M-RET)
$\frac{\Gamma(x) = s}{\Gamma \vdash x : s}$	(M-VAR)	$\frac{\Gamma \vdash e_1 : \mathbb{M} a' \quad \Gamma, x : a' \vdash e_2 : \mathbb{M} a}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \mathbb{M} a}$	(M-LET)
$\frac{\text{arity}(\text{op}) = \tilde{e} \quad \Gamma \vdash \tilde{e} : \widetilde{\text{int}}}{\Gamma \vdash \text{op}(\tilde{e}) : \text{int}}$	(M-OP)	$\frac{}{\Gamma \vdash \text{call}^{act} : t^{act}}$	(M-ACT)
$\frac{\Gamma, x : s \vdash e : r}{\Gamma \vdash \lambda x : s. e : s \rightarrow r}$	(M-ABS)		
$\frac{\text{arity}(\text{pred}) = \tilde{e}' \quad \Gamma \vdash \tilde{e}' : \widetilde{\text{int}}}{\Gamma \vdash \text{if pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 : \mathbb{M} a}$	(M-IF)	$\frac{\Gamma \vdash e_1 : \mathbb{M} a \quad \Gamma \vdash e_2 : \mathbb{M} a}{\Gamma \vdash \text{if pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 : \mathbb{M} a}$	

Fig. 1. Typing rules of the language

We have so far parameterized the language by effects. A concrete language and its semantics are determined by the parameters given below, which we call *effect parameters*. The semantics of effect operations are given in the form of predicate transformers; we assume the simply-typed lambda calculus with a and prop (which is the type of propositions) as base types and also with appropriate constants (such as logical connectives $\wedge, \vee : \text{prop} \rightarrow \text{prop} \rightarrow \text{prop}$, the predicate $\text{pred} : \text{int}^k \rightarrow \text{prop}$ for each k -ary predicate pred in the source language, and the operator $\text{op} : \text{int}^k \rightarrow \text{int}$ for each k -ary operator op in the source language) for describing predicate transformers.

- $\mathbf{W}$: The type operator that takes a base type a and returns the (simple) type of the predicate transformer for computations returning values of type a .
- $\mathbf{return}^{\mathbf{W}} : \forall a. a \rightarrow \mathbf{W} a$: The predicate transformer of a computation that takes a value of type a and just returns it without any effects.
- $\mathbf{bind}^{\mathbf{W}} : \forall a_1, a. (\mathbf{W} a_1 \rightarrow (a_1 \rightarrow \mathbf{W} a) \rightarrow \mathbf{W} a)$: The predicate transformer of a computation that takes w_1 and w_2 of types $\mathbf{W} a_1$ and $a_1 \rightarrow \mathbf{W} a$ respectively, and pipes the result of the computation specified by w_1 to the computation specified by w_2 .
- $\mathbf{A}$: A set of (names of) primitive effect actions. We use the metavariable act to range over this set.
- t^{act} : The (monadic) type of each action $act \in \mathbf{A}$. This describes the types of parameters and return values of each action.
- w^{act} : The predicate transformer of each action $act \in \mathbf{A}$. It describes the behavior of each action. The simple type of w^{act} must be consistent with t^{act} , in the sense that w^{act} must have the type obtained from t^{act} by replacing $\mathbf{M}$ with $\mathbf{W}$. For example, if $t^{act} = \mathbf{unit} \rightarrow \mathbf{M} \mathbf{int}$, then w^{act} must have type $\mathbf{unit} \rightarrow \mathbf{W} \mathbf{int}$.

In the field of the DM approach, the former three parameters are called a specification monad, and the latter three yield the DM type of effect actions. Methods to obtain these parameters in a correct-by-construction manner have been studied by Ahman et al. [1] and Maillard et al. [12]. According to Ahman et al.’s result, reinterpreted by Maillard et al., for effects that can be represented as monad transformers, these parameters can be obtained by applying the monad transformer to the continuation monad with $\mathbf{prop}$ as the answer type.

We give examples of effect parameters, some of which can be derived from monad transformers.

Example 1. The simplest case is the purely functional one. In this case, there are no effect actions: $\mathbf{A} = \emptyset$. The rest of the effect parameters can be defined as:

- $\mathbf{W} a := (a \rightarrow \mathbf{prop}) \rightarrow \mathbf{prop}$
- $\mathbf{return}^{\mathbf{W}} (x : a) := \lambda p : (a \rightarrow \mathbf{prop}). p \ x$
- $\mathbf{bind}^{\mathbf{W}} (w_1 : \mathbf{W} a_1) (w_2 : a_1 \rightarrow \mathbf{W} a) := \lambda p : (a \rightarrow \mathbf{prop}). w_1 (\lambda x. w_2 \ x \ p)$

These coincide with the identity monad transformer applied to the continuation monad.

Example 2. For programs with a state of type $\mathbf{int}$, the effect parameters are:⁷

- $\mathbf{W} a := \mathbf{int} \rightarrow (a \rightarrow \mathbf{int} \rightarrow \mathbf{prop}) \rightarrow \mathbf{prop}$
- $\mathbf{return}^{\mathbf{W}} (x : a) := \lambda h : \mathbf{int}. \lambda p : (a \rightarrow \mathbf{int} \rightarrow \mathbf{prop}). p \ x \ h$
- $\mathbf{bind}^{\mathbf{W}} (w_1 : \mathbf{W} a_1) (w_2 : a_1 \rightarrow \mathbf{W} a)$
 $\quad := \lambda h : \mathbf{int}. \lambda p : (a_2 \rightarrow \mathbf{int} \rightarrow \mathbf{prop}). w_1 \ h \ (\lambda x : a_2. \lambda h' : \mathbf{int}. w_2 \ x \ h' \ p)$
- $\mathbf{A} = \{get, put\}$
 - $t^{get} := \mathbf{unit} \rightarrow \mathbf{M} \mathbf{int}$

⁷ Usually, $\mathbf{W} a$ is defined as $S \rightarrow (a \times S \rightarrow \mathbf{prop}) \rightarrow \mathbf{prop}$ for a store type S . To make the language minimum, however we avoid using pairs by using the currying transformation.

- $w^{get} := \lambda().\lambda h : \text{int}.\lambda p : (\text{int} \rightarrow \text{int} \rightarrow \text{prop}).p \ h \ h$
- $t^{put} := \text{int} \rightarrow \mathbf{M} \ \text{unit}$
- $w^{put} := \lambda x : \text{int}.\lambda h : \text{int}.\lambda p : (\text{unit} \rightarrow \text{int} \rightarrow \text{prop}).p \ () \ x$

These parameters can be obtained by applying the state monad transformer to the continuation monad. Note that w^{get} and w^{put} have types $\text{unit} \rightarrow \mathbf{W} \ \text{int}$ and $\text{int} \rightarrow \mathbf{W} \ \text{unit}$ respectively, obtained from t^{get} and t^{put} by replacing $\mathbf{M}$ with $\mathbf{W}$.

Given the effect parameters above, the semantics of the language can be given as a predicate transformer semantics. We omit the definition here, as the translation from programs to HFL(Z) formulas given in Section 3.3 can actually be viewed as the predicate transformer semantics (although the authors [9,17] did not provide that observation).

3.2 The Dijkstra Monad Approach

The Dijkstra monad approach is formalized as another type system for the language, which we call the DM type system. We use $\vdash^{\text{DM}}$ instead of $\vdash$ for a type judgment of the DM type system. The syntax of types is given below.

a (base types) $::= \text{unit} \mid \text{int}$
 π (predicate transformer types) $::= a \mid \text{prop} \mid \pi \rightarrow \pi$
 τ (function types) $::= x : \varsigma \rightarrow \rho \mid \forall u : \pi. \tau$
 ς (argument types) $::= a \mid \tau$ ρ (return types) $::= \mathbf{D} \ a \ w \mid \tau$
 w (predicate transformers) $::= e \mid \text{pred}(\tilde{e}) \mid \neg \text{pred}(\tilde{e}) \mid w_1 \wedge w_2 \mid w_1 \vee w_2$
 $\mid \lambda u : \pi. w \mid w_1 \ w_2 \mid \text{if } \text{pred}(\tilde{e}) \text{ then } w_1 \text{ else } w_2$

Compared with the simple monadic type system, we introduced the type π for predicate transformers, and changed the type of a monadic computation from $\mathbf{M} \ a$ to $\mathbf{D} \ a \ w$, where w is a predicate transformer of type $\mathbf{W} \ a$. The metavariable a is inherited from the simple type system, and the metavariables τ, ς , and ρ correspond to t, s , and r respectively, in the simple monadic type system. Function types have been extended to dependent function types of the form $x : a_1 \rightarrow \mathbf{D} \ a_2 \ w$, etc., so that x may occur in the predicate transformer w . We also allow polymorphic types $\forall u : \pi. \tau$ on predicate transformers u . The language of predicate transformers, ranged over by w , is defined by extending the expression syntax with constructs of predicate logic, λ -abstractions and applications of the type π , and (non-monadic) conditional expressions; we assume that the language of predicate transformers subsumes the simply-typed λ -calculus assumed in Section 3.1 for describing predicate transformers. We may omit type annotations for predicate transformer variables and just write $\lambda u. w$ and $\forall u. \tau$ for $\lambda u : \pi. w$ and $\forall u : \pi. \tau$ when they are clear from the context. Note that this definition is a fragment of the original type system of Dijkstra monads [15]: for example, refinement types and polymorphism are excluded.

The typing rules of the DM type system for expressions and predicate transformers are shown in Figures 2 and 3, respectively. In the rule DM-ABS, ς^{toM}

$\frac{}{\Gamma \vdash^{\text{DM}} () : \text{unit}}$	(DM-UNIT)	$\frac{\Gamma \vdash^{\text{DM}} e : a}{\Gamma \vdash^{\text{DM}} \text{return } e : \mathbb{D} a (\text{return}^{\mathbb{W}} e)}$	(DM-RET)
$\frac{}{\Gamma \vdash^{\text{DM}} n : \text{int}}$	(DM-INT)	$\frac{\Gamma \vdash^{\text{DM}} e : \tau \quad \Gamma \vdash^{\text{DM}} \tau =_{\beta\eta} \tau'}{\Gamma \vdash^{\text{DM}} e : \tau'}$	(DM-CONV)
$\frac{\Gamma(x) = \varsigma}{\Gamma \vdash^{\text{DM}} x : \varsigma}$	(DM-VAR)	$\frac{\Gamma, u : \pi \vdash^{\text{DM}} e : \tau}{\Gamma \vdash^{\text{DM}} e : \forall u : \pi. \tau}$	(DM- $\forall$ -I)
$\frac{\text{arity}(\text{op}) = \tilde{e} \quad \Gamma \vdash^{\text{DM}} \tilde{e} : \widetilde{\text{int}}}{\Gamma \vdash^{\text{DM}} \text{op}(\tilde{e}) : \text{int}}$	(DM-OP)	$\frac{\Gamma \vdash^{\text{DM}} e : \forall u : \pi. \tau \quad \Gamma \vdash^{\text{DM}} w : \pi}{\Gamma \vdash^{\text{DM}} e : [w/u]\tau}$	(DM- $\forall$ -E)
$\frac{\Gamma, x : \varsigma \vdash^{\text{DM}} e : \rho \quad \varsigma^{\text{toM}} = s}{\Gamma \vdash^{\text{DM}} \lambda x : s. e : x : \varsigma \rightarrow \rho}$	(DM-ABS)	$\frac{}{\Gamma \vdash^{\text{DM}} \text{call}^{\text{act}} : \text{toDM}[t^{\text{act}}, w^{\text{act}}]}$	(DM-ACT)
$\frac{\Gamma \vdash^{\text{DM}} e_1 : x : \varsigma \rightarrow \rho \quad \Gamma \vdash^{\text{DM}} e_2 : \varsigma}{\Gamma \vdash^{\text{DM}} e_1 e_2 : ([e_2/x]\rho)}$	(DM-APP)	$\frac{\text{arity}(\text{pred}) = \tilde{e}' \quad \Gamma \vdash^{\text{DM}} \tilde{e}' : \text{int} \quad \Gamma \vdash^{\text{DM}} e_1 : \mathbb{D} a w_1 \quad \Gamma \vdash^{\text{DM}} e_2 : \mathbb{D} a w_2}{\Gamma \vdash^{\text{DM}} \text{if pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 : \mathbb{D} a (\text{if pred}(\tilde{e}') \text{ then } w_1 \text{ else } w_2)}$	(DM-IF)
		$\frac{\Gamma \vdash^{\text{DM}} e_1 : \mathbb{D} a' w_1 \quad \Gamma, x : a' \vdash^{\text{DM}} e_2 : \mathbb{D} a w_2}{\Gamma \vdash^{\text{DM}} \text{let } x = e_1 \text{ in } e_2 : \mathbb{D} a (\text{bind}^{\mathbb{W}} w_1 (\lambda x. w_2))}$	(DM-LET)

Fig. 2. Typing rules of DM for expressions

denotes the simple monadic type obtained by erasing predicate transformers, as defined by:

$$\begin{aligned}
 a^{\text{toM}} &:= a \quad (\mathbb{D} a w)^{\text{toM}} = \mathbb{M} a \\
 (x : \varsigma \rightarrow \rho)^{\text{toM}} &:= \varsigma^{\text{toM}} \rightarrow \rho^{\text{toM}} \quad (\forall u : \pi. \tau)^{\text{toM}} := \tau^{\text{toM}}.
 \end{aligned}$$

The rules DM-RET and DM-LETIN reveal that the Dijkstra monad $\mathbb{D} a w$ is a monad indexed by the monad $\mathbb{W}$ of predicate transformers. The monadic operators $\text{return}^{\mathbb{W}}$ and $\text{bind}^{\mathbb{W}}$ of $\mathbb{W}$ are used in the typing rules of return expressions and let-in expressions, respectively. When a pure expression is lifted to an effectful computation using the return expression, it is injected into the predicate transformer using $\text{return}^{\mathbb{W}}$. The predicate transformer of a let-expression is computed by composing the two predicate transformers of its two sub-expressions using $\text{bind}^{\mathbb{W}}$.

In addition to typing rules guided by the expression syntax, we have rules DM-CONV, DM- $\forall$ -I, and DM- $\forall$ -E for conversion and introduction/elimination of universal quantifiers. The original type system of Dijkstra monads [15] has a subsumption rule, but we introduced the conversion rule instead for simplicity. The equivalence relation $\Gamma \vdash^{\text{DM}} \varsigma =_{\beta\eta} \varsigma'$ is defined based on the $\beta\eta$ -equality of predicate transformers inductively on the structure of types, where the base case $\Gamma \vdash^{\text{DM}} \mathbb{D} a w_1 =_{\beta\eta} \mathbb{D} a w_2$ holds just if $w_1 =_{\beta\eta} w_2$. The type of an action call^{act} is constructed from the simple type t^{act} and the predicate transformer w^{act} , using the transformation $\text{toDM}[r, w]$, proposed by Ahman et al. [1]. Intuitively, this

$$\begin{array}{c}
\frac{\Gamma(x) = \pi}{\Gamma \vdash^{\text{DM}} x : \pi} \quad (\text{DM-VAR}^\pi) \quad \frac{\text{arity}(\mathbf{pred}) = |\tilde{e}| \quad \Gamma \vdash^{\text{DM}} \tilde{e} : \widetilde{\mathbf{int}}}{\Gamma \vdash^{\text{DM}} \neg \mathbf{pred}(\tilde{e}) : \mathbf{prop}} \quad (\text{DM-NPR}) \\
\frac{\Gamma, x : \pi_1 \vdash^{\text{DM}} w : \pi_2}{\Gamma \vdash^{\text{DM}} \lambda x : \pi_1. w : \pi_1 \rightarrow \pi_2} \quad (\text{DM-ABS}^\pi) \quad \frac{\Gamma \vdash^{\text{DM}} w_1 : \mathbf{prop} \quad \Gamma \vdash^{\text{DM}} w_2 : \mathbf{prop}}{\Gamma \vdash^{\text{DM}} w_1 \wedge w_2 : \mathbf{prop}} \quad (\text{DM-}\wedge) \\
\frac{\Gamma \vdash^{\text{DM}} w_1 : \pi_1 \rightarrow \pi_2 \quad \Gamma \vdash^{\text{DM}} w_2 : \pi_1}{\Gamma \vdash^{\text{DM}} w_1 w_2 : \pi_2} \quad (\text{DM-APP}^\pi) \quad \frac{\Gamma \vdash^{\text{DM}} w_1 : \mathbf{prop} \quad \Gamma \vdash^{\text{DM}} w_2 : \mathbf{prop}}{\Gamma \vdash^{\text{DM}} w_1 \vee w_2 : \mathbf{prop}} \quad (\text{DM-V}) \\
\frac{\text{arity}(\mathbf{pred}) = |\tilde{e}| \quad \Gamma \vdash^{\text{DM}} \tilde{e} : \widetilde{\mathbf{int}}}{\Gamma \vdash^{\text{DM}} \mathbf{pred}(\tilde{e}) : \mathbf{prop}} \quad (\text{DM-PR}) \\
\frac{\text{arity}(\mathbf{pred}) = |\tilde{e}| \quad \Gamma \vdash^{\text{DM}} \tilde{e} : \widetilde{\mathbf{int}} \quad \Gamma \vdash^{\text{DM}} w_1 : \pi \quad \Gamma \vdash^{\text{DM}} w_2 : \pi}{\Gamma \vdash^{\text{DM}} \mathbf{if } \mathbf{pred}(\tilde{e}) \mathbf{ then } w_1 \mathbf{ else } w_2 : \pi} \quad (\text{DM-IF}^\pi)
\end{array}$$

Fig. 3. Typing rules of DM for predicate transformers

transformation yields a DM type of an expression of type r whose behavior is specified by the predicate transformer w . It is defined as follows.

$$\text{toDM}[M a, w] := D a w$$

$$\text{toDM}[a \rightarrow r, w] := x : a \rightarrow \text{toDM}[r, w x]$$

$$\text{toDM}[t \rightarrow r, w] := \forall w_x : t^{\text{WP}}. x : \text{toDM}[t, w_x] \rightarrow \text{toDM}[r, w w_x] \quad (x, w_x : \text{fresh}).$$

As we see in the third line, when r is an arrow type taking a function as the parameter x , we universally quantify over the predicate transformer w_x of the function. The type of w_x is calculated from the simple type r , defined below:

$$a^{\text{WP}} := a \quad (M a)^{\text{WP}} := W a \quad (s \rightarrow r)^{\text{WP}} := s^{\text{WP}} \rightarrow r^{\text{WP}}$$

Essentially, the translation replaces the monadic type $M a$ with the type $W a$. Note that the transformations $\text{toDM}[r, w]$ and r^{WP} are called *elaboration* and the $\star$ -*transformation*, respectively, by Ahman et al. [1]. Below we demonstrate how the DM type of call^{put} is calculated. Recall the definitions $t^{\text{put}} = \mathbf{int} \rightarrow M \mathbf{unit}$ and $w^{\text{put}} = \lambda x. \lambda h. \lambda p. p () x$ from Example 2.

$$\begin{aligned}
\text{toDM}[t^{\text{put}}, w^{\text{put}}] &= \text{toDM}[\mathbf{int} \rightarrow M \mathbf{unit}, \lambda x. \lambda h. \lambda p. p () x] \\
&= x : \mathbf{int} \rightarrow D \mathbf{unit} ((\lambda x. \lambda h. \lambda p. p () x) x) \\
&=_{\beta} x : \mathbf{int} \rightarrow D \mathbf{unit} (\lambda h. \lambda p. p () x)
\end{aligned}$$

A higher-order example of the calculation $\text{toDM}[t, w]$ is given later in Section 3.4. Note that the DM type $\text{toDM}[r, w]$ is consistent with the simple type r , that is, $(\text{toDM}[r, w])^{\text{toM}} = r$.

Example 3. Let us consider the following expression, which is equivalent to the function `twice` discussed in 2.3.

$$\lambda f : (\mathbf{int} \rightarrow M \mathbf{int}). \lambda x : \mathbf{int}. \mathbf{let } y = f x \mathbf{ in } f y.$$

This expression is typed as follows:

$$\begin{array}{c}
\dfrac{\dots}{\Gamma \vdash^{\text{DM}} f x : \text{D int } (w_f x)} \quad \dfrac{\dots}{\Gamma, y : \text{int} \vdash^{\text{DM}} f y : \text{D int } (w_f y)} \quad \text{DM-LetIn} \\
\dfrac{\Gamma \vdash^{\text{DM}} \text{let } y = f x \text{ in } f y : \text{D int } (\text{bind}^{\text{W}}(w_f x) (\lambda y. w_f y))}{w_f : \pi_{w_f}, f : \tau_f \vdash^{\text{DM}} \lambda x : \text{int}. e_{\text{twice}} : x : \text{int} \rightarrow \text{D int } w_{\text{twice}}} \quad \text{DM-Abs} \\
\dfrac{\Gamma \vdash^{\text{DM}} \lambda f : (\text{int} \rightarrow \text{M int}). \lambda x : \text{int}. e_{\text{twice}} : f : \tau_f \rightarrow x : \text{int} \rightarrow \text{D int } w_{\text{twice}}}{\vdash^{\text{DM}} \lambda f : (\text{int} \rightarrow \text{M int}). \lambda x : \text{int}. e_{\text{twice}} : \forall w_f. f : \tau_f \rightarrow x : \text{int} \rightarrow \text{D int } w_{\text{twice}}} \quad \text{DM-}\forall\text{-I}
\end{array}$$

Here, $\Gamma := w_f : \pi_{w_f}, f : \tau_f, x : \text{int}$, where:

$$\begin{array}{ll}
\pi_{w_f} := \text{int} \rightarrow \text{W int} & \tau_f := z : \text{int} \rightarrow \text{D int } (w_f z) \\
w_{\text{twice}} := \text{bind}^{\text{W}}(w_f x) (\lambda y. w_f y) & e_{\text{twice}} := \text{let } y = f x \text{ in } f y
\end{array}$$

3.3 The HFL(Z) Approach

This subsection formalizes a translation from a program to a HFL(Z) formula, which represents the predicate transformer of the program. HFL(Z) is a higher-order logic with least/greatest fixpoints and integers, introduced by Kobayashi et al. [9] as an extension of higher-order modal fixpoint logic [16].

Syntax The syntax of HFL(Z) formulas is defined by:

$$\begin{array}{l}
\kappa \text{ (types)} ::= \text{prop} \mid \eta \rightarrow \kappa \quad \eta ::= \text{unit} \mid \text{int} \mid \kappa \\
\varphi \text{ (formulas)} ::= () \mid n \mid x \mid \text{op}(\tilde{\varphi}) \mid \text{pred}(\tilde{\varphi}) \mid \neg \text{pred}(\tilde{\varphi}) \\
\quad \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \lambda x : \eta. \varphi \mid \varphi_1 \varphi_2 \mid \mu x : \kappa. \varphi \mid \nu x : \kappa. \varphi.
\end{array}$$

In addition to ordinary constructs of predicate logic, the syntax includes λ -abstractions and applications for manipulating higher-order predicates, and the least/greatest fixpoint operators μ and ν . The formulas $\mu x : \kappa. \varphi$ and $\nu x : \kappa. \varphi$ respectively denote the least and greatest predicates of type κ such that $x \equiv \varphi$. We sometimes use a fixpoint equation $x : \kappa =_{\alpha} \varphi$ (where $\alpha \in \{\mu, \nu\}$) to define x as the predicate $\alpha x : \kappa. \varphi$. Actually, fixpoint formulas are required only for encoding recursive functions; thus, we do not use them in this section.

HFL(Z) [9] has an associated simple type system to exclude ill-formed formulas. The type system is essentially that of the simply-typed λ -calculus with operators, constant predicates, and logical connectives. A concrete definition of the type system is deferred to the longer version [19]. We assume that HFL(Z) contains, as a sublanguage, the simply-typed λ -calculus assumed in Section 3.1 for describing predicate transformers.

Reduction from Program Verification In this section, we introduce the translation from programs to HFL(Z) formulas.

We first define the translation $(\cdot)^{\text{HFL}}$ for types by:

$$a^{\text{HFL}} := a \quad (\text{M } a)^{\text{HFL}} := \text{W } a \quad (s \rightarrow r)^{\text{HFL}} := s^{\text{HFL}} \rightarrow r^{\text{HFL}}.$$

Actually, this transformation is the same as $(\cdot)^{\text{WP}}$, introduced in the DM type system for computing the type of predicate transformers from a simple monadic type. Thus, we use the notation r^{WP} instead of r^{HFL} hereafter.

The translation of expressions is defined by:⁸

$$\begin{aligned}
()^{\text{HFL}} &:= () & n^{\text{HFL}} &:= n & x^{\text{HFL}} &:= x & (\lambda x : s. e)^{\text{HFL}} &:= \lambda x : s^{\text{WP}}. e^{\text{HFL}} \\
(\text{op}(\tilde{e}))^{\text{HFL}} &:= \text{op}(\tilde{e}^{\text{HFL}}) & (e_1 e_2)^{\text{HFL}} &:= e_1^{\text{HFL}} e_2^{\text{HFL}} \\
\left(\text{if } \text{pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 \right)^{\text{HFL}} & \\
&:= \text{IfThenElse}[\mathbb{W} a, \text{pred}(\tilde{e}'^{\text{HFL}}), e_1^{\text{HFL}}, e_2^{\text{HFL}}] \\
&\quad (\text{when } \Gamma \vdash \text{if } \text{pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 : \mathbb{M} a) \\
(\text{return } e)^{\text{HFL}} &:= \text{return}^{\mathbb{W}}(e^{\text{HFL}}) & (\text{call}^{\text{act}})^{\text{HFL}} &:= w^{\text{act}} \\
(\text{let } x = e_1 \text{ in } e_2)^{\text{HFL}} &:= \text{bind}^{\mathbb{W}}(e_1^{\text{HFL}})(\lambda x. e_2^{\text{HFL}}).
\end{aligned}$$

Here, $\text{IfThenElse}[\kappa, \text{pred}(\tilde{e}'), e_1, e_2]$ is a macro introduced for the convenience of stating the correspondence between the DM and HFL(Z) approaches in Section 3.4. It is expanded as follows.

$$\begin{aligned}
\text{IfThenElse}[\text{prop}, \text{pred}(\tilde{e}'), e_1, e_2] &= (\text{pred}(\tilde{e}') \wedge e_1) \vee (\neg \text{pred}(\tilde{e}') \wedge e_2) \\
\text{IfThenElse}[\eta \rightarrow \kappa, \text{pred}(\tilde{e}'), e_1, e_2] &= \lambda x : \eta. \text{IfThenElse}[\kappa, \text{pred}(\tilde{e}'), e_1 x, e_2 x].
\end{aligned}$$

The return and bind expressions are encoded using the operators of the monad $\mathbb{W}$. In the case of the pure computation monad, the effect of (selective) CPS translation is achieved through $\text{return}^{\mathbb{W}}$ and $\text{bind}^{\mathbb{W}}$.

Example 4. Let us consider the function `twice again` as an example:

$$\lambda f : (\text{int} \rightarrow \mathbb{M} \text{int}). \lambda x : \text{int}. \text{let } y = f x \text{ in } f y.$$

It is translated to:

$$\begin{aligned}
&(\lambda f : (\text{int} \rightarrow \mathbb{M} \text{int}). \lambda x : \text{int}. \text{let } y = f x \text{ in } f y)^{\text{HFL}} \\
&= \lambda f : \pi_f. \lambda x : \text{int}. \text{bind}^{\mathbb{W}}(f x)(\lambda y : \text{int}. f y).
\end{aligned}$$

Here, π_f abbreviates $(\text{int} \rightarrow \mathbb{M} \text{int})^{\text{WP}}$, which equals $\text{int} \rightarrow \mathbb{W} \text{int}$. □

Remark 1. Kobayashi et al. [9,17] only sketched how to deal with side effects (other than “events” [17]) in an example-based manner. Thus, the general translation given above using effect monads is also a contribution of this paper. □

⁸ Here, the actual translation should take a type judgment. For the sake of simplicity, we refer to the type environment and the type of an expression only where necessary.

3.4 Formalization of the Correspondence

Now we are ready to formalize the correspondence between the two approaches. We first show that there is a type derivation in the Dijkstra monad approach corresponding to the program transformation of the HFL(Z) approach.

The type system of the Dijkstra monad approach allows freedom in the choice of types of function parameters and the use of conversion and quantification rules. For example, $\lambda f : (\text{int} \rightarrow \mathbf{M} \text{ int}). \lambda x : \text{int}. f(x + 1)$, which has type $(\text{int} \rightarrow \mathbf{M} \text{ int}) \rightarrow \text{int} \rightarrow \mathbf{M} \text{ int}$ in the simple monadic type system, can be given any of the following types in the DM type system:

$$\begin{aligned} \forall w_f. f : (y : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f y)) &\rightarrow x : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f(x + 1)) \\ f : (y : \text{int} \rightarrow \mathbf{D} \text{ int } \lambda p. p(y + 10)) &\rightarrow x : \text{int} \rightarrow \mathbf{D} \text{ int } (\lambda p. p(x + 11)) \\ f : (y : \text{int} \rightarrow \mathbf{D} \text{ int } \lambda p. p(y)) &\rightarrow x : \text{int} \rightarrow \mathbf{D} \text{ int } (\lambda p. p(x + 1)). \end{aligned}$$

The first type abstracts over the predicate transformer of f , while the second and third types accept only function arguments having a specific predicate transformer. For example, a function of the first type can be applied to both $\lambda y.\text{return}(y + 10)$ and $\lambda y.\text{return } y$, but a function of the second type can be applied to $\lambda y.\text{return}(y + 10)$ but not to $\lambda y.\text{return } y$, and a function of the third type can be applied to $\lambda x.\text{return } y$ but not to $\lambda y.\text{return}(y + 10)$. Since the output of the transformation by the HFL(Z) approach is $\lambda f. \lambda x. f(x + 1)$, deriving the first type

$$\forall w_f. f : (y : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f y)) \rightarrow x : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f(x + 1))$$

is the most natural choice in order to show the correspondence. Actually, this type is obtained as $\text{toDM}[(\text{int} \rightarrow \mathbf{M} \text{ int}) \rightarrow \text{int} \rightarrow \mathbf{M} \text{ int}, \lambda f. \lambda x. f(x + 1)]$, as confirmed by the following calculation.

$$\begin{aligned} &\text{toDM}[(\text{int} \rightarrow \mathbf{M} \text{ int}) \rightarrow \text{int} \rightarrow \mathbf{M} \text{ int}, \lambda f. \lambda x. f(x + 1)] \\ &= \forall w_f. f : \text{toDM}[\text{int} \rightarrow \mathbf{M} \text{ int}, w_f] \rightarrow \text{toDM}[\text{int} \rightarrow \mathbf{M} \text{ int}, (\lambda f. \lambda x. f(x + 1))w_f] \\ &= \dots \\ &= \forall w_f. f : (y : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f y)) \rightarrow x : \text{int} \rightarrow \mathbf{D} \text{ int } (w_f(x + 1)). \end{aligned}$$

Here, we have omitted the type of w_f . Actually, this observation can be generalized: when the parameter f has the simple type t , we choose $\text{toDM}[t, w_f]$ as the DM type for f , where w_f is universally quantified. Hereafter, we call $\text{toDM}[t, w_f]$ as the canonical DM type corresponding to the simple type t specified by the predicate transformer w_f .

We also have to bridge the gap on the treatment of conditional expressions: **if** $\text{pred}(\tilde{e}')$ **then** e_1 **else** e_2 . The DM type system derives the predicate transformer of this expression as **if** $\text{pred}(\tilde{e}')$ **then** w_1 **else** w_2 , where w_1 and w_2 are predicate transformers of e_1 and e_2 , respectively. On the other hand, HFL(Z) formulas do not include conditional expressions, so they must be encoded as formulas, using the macro $\text{IfThenElse}[w a, \text{pred}(\tilde{e}'), w_1, w_2]$. We bridge this gap by

the translation $w^\dagger$, which replaces conditional expressions in DM predicate transformers with the corresponding formula using this macro, defined as follows⁹.

$$\begin{aligned} (\text{if } \text{pred}(\tilde{e}) \text{ then } w_1 \text{ else } w_2)^\dagger &:= \text{IfThenElse}[\pi, \text{pred}(\tilde{e}^\dagger), w_1^\dagger, w_2^\dagger] \\ (\text{when } \Gamma \vdash^{\text{DM}} \text{if } \text{pred}(\tilde{e}) \text{ then } w_1 \text{ else } w_2 : \pi) \end{aligned}$$

The following theorem formalizes the syntactic correspondence between the DM type system and the program transformation of the HFL(Z) approach.

Theorem 1. *If $\vdash e : r$, then there exists a predicate transformer w such that $\vdash^{\text{DM}} e : \text{toDM}[r, w]$ and $w^\dagger = e^{\text{HFL}}$.*

Example 5. Suppose the effect parameters are the ones for stateful computation given in Example 2. Let e be the expression:

`let  $x = \text{get}()$  in if  $x > 0$  then return  $x$  else return 0.`

It is transformed to the following HFL(Z) formula:

$$\begin{aligned} e^{\text{HFL}} &= \text{bind}^W (w^{\text{get}}()) (\lambda x. \text{IfThenElse}[W \text{int}, x > 0, \text{return}^W x, \text{return}^W 0]) \\ &= \text{bind}^W (w^{\text{get}}()) (\lambda x. \lambda h. \lambda p. x > 0 \wedge \text{return}^W x \ h \ p \vee x \leq 0 \wedge \text{return}^W 0 \ h \ p)). \end{aligned}$$

Let w be the predicate transformer:

$$w := \text{bind}^W (w^{\text{get}}()) (\lambda x. \text{if } x > 0 \text{ then return}^W x \text{ else return}^W 0).$$

This w satisfies $\vdash^{\text{DM}} e : \text{D int } w$ and $w^\dagger = e^{\text{HFL}}$. □

To prove the theorem above, we introduce another transformation e^{DMWP} , which serves as a witness for w in the theorem. It is the same as $(\cdot)^{\text{HFL}}$, except the following case for conditional expressions, which are transformed into those of predicate transformers (because conditional expressions are available in the DM language for predicate transformers).

$$\begin{aligned} & \left(\text{if } \text{pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 \right)^{\text{DMWP}} \\ &:= \text{if } \text{pred}(\widetilde{e'^{\text{DMWP}}}) \text{ then } e_1^{\text{DMWP}} \text{ else } e_2^{\text{DMWP}} \\ & \quad (\text{when } \Gamma \vdash \text{if } \text{pred}(\tilde{e}') \text{ then } e_1 \text{ else } e_2 : \text{M } a) \end{aligned}$$

Next, we show that e^{DMWP} is indeed a witness for w in the theorem. To show that $w = e^{\text{DMWP}}$ implies $\vdash^{\text{DM}} e : \text{toDM}[r, w]$, we need to generalize this

⁹ For other cases, we just compositionally apply the translation. For example, $(e_1 e_2)^\dagger := e_1^\dagger e_2^\dagger$. Note that this translation is not always guaranteed to yield well-formed HFL(Z) formulas. Nevertheless, this is not a problem for the correspondence formalized as Theorem 1, as this theorem claims the *existence* of a DM predicate transformer that corresponds to a HFL(Z) formula via this translation.

property to open terms. To that end, we define the “canonical” type environment of the DM type system that corresponds to a type environment of the simple monadic type system, as follows.

$$\begin{aligned} \emptyset^{\text{DM}} &:= \emptyset & (\Gamma, x : a)^{\text{DM}} &:= \Gamma^{\text{DM}}, x : a \\ (\Gamma, x : t)^{\text{DM}} &:= \Gamma^{\text{DM}}, w_x : t^{\text{WP}}, x : \text{toDM}[t, w_x] & (\text{where } w_x : \text{fresh}) \end{aligned}$$

Here, variables of a base type a are kept as they are, while a variable of type t is replaced by a pair of w_x and x , where x has the canonical type specified by w_x .

To deal with open terms, we also need to rename variables. This is due to the observation that the DM type abstracts over the behavior of functional parameters using universal quantification of its predicate transformer, while the result of the transformation $(\cdot)^{\text{DMWP}}$ takes the translated higher-order predicate as an argument. For example, the function $\lambda f : (\text{int} \rightarrow \mathbf{M} \text{ int}). f \ 0$ can be given the type $\forall w_f. f : (x : \text{int} \rightarrow \mathbf{D} \text{ int} (w_f \ x)) \rightarrow \mathbf{D} \text{ int} (w_f \ 0)$ while the program transformation yields $\lambda f. f \ 0$. As indicated by the red color, f in $\lambda f. f \ 0$ actually corresponds to w_f in the DM type, rather than f . Based on this observation, we rename f to w_f , by using the substitution σ_Γ , defined by: $\sigma_\Gamma(x) = w_x$ if $\Gamma(x)$ contains $\mathbf{M}$ and $\sigma_\Gamma(x) = x$ otherwise.

Now, the correspondence between the typing rules of DM and the program transformation $(\cdot)^{\text{DMWP}}$ on open terms can be stated as the following lemma.

- Lemma 1.** 1. If $\Gamma \vdash e : a$, then $\Gamma^{\text{DM}} \vdash^{\text{DM}} e : a$ and $\sigma_\Gamma(e^{\text{DMWP}}) = e$.
 2. If $\Gamma \vdash e : \mathbf{M} \ a$, then $\Gamma^{\text{DM}} \vdash^{\text{DM}} e : \mathbf{D} \ a \ \sigma_\Gamma(e^{\text{DMWP}})$, and $\Gamma^{\text{DM}} \vdash^{\text{DM}} \sigma_\Gamma(e^{\text{DMWP}}) : \mathbf{W} \ a$.
 3. If $\Gamma \vdash e : t$, then $\Gamma^{\text{DM}} \vdash^{\text{DM}} e : \text{toDM}[t, \sigma_\Gamma(e^{\text{DMWP}})]$ and $\Gamma^{\text{DM}} \vdash^{\text{DM}} \sigma_\Gamma(e^{\text{DMWP}}) : t^{\text{WP}}$.

The proof is found in the longer version [19]. As a special case of the lemma above, when the type environment Γ is empty, we have $\vdash^{\text{DM}} e : \text{toDM}[r, w]$ for $w = e^{\text{DMWP}}$, confirming that $w = e^{\text{DMWP}}$ indeed serves as a witness for the condition $\vdash^{\text{DM}} e : \text{toDM}[r, w]$ of Theorem 1. It remains to show $w = e^{\text{DMWP}}$ also satisfies $w^\dagger = e^{\text{HFL}}$, the second condition of Theorem 1, as stated in the following lemma.

- Lemma 2.** If $\Gamma \vdash e : r$, then $e^{\text{DMWP}\dagger} = e^{\text{HFL}}$.

We now obtain Theorem 1 as a corollary of the two lemmas above.

Proof (Theorem 1). Let $w = e^{\text{DMWP}}$. Then by the special cases of Lemmas 1 and 2 where Γ is empty, we have $\vdash^{\text{DM}} e : \text{toDM}[r, w]$ and $w^\dagger = e^{\text{HFL}}$.

We now show the converse, i.e., that for every predicate transformer derived in the DM approach, an at least equally good predicate transformer can be obtained in the HFL(Z) approach. We assume that the DM type system is sound with respect to the predicate transformer semantics (as defined in Section 3.3 by the transformation to HFL(Z) formulas), i.e., if $\vdash^{\text{DM}} e : \mathbf{D} \ a \ w$ holds, then w is a sound predicate transformer (even though it may not be the weakest one). Then the following theorem follows immediately from the assumption.

Theorem 2. *Suppose $\vdash e : \mathbf{M} a$ and $\vdash^{\text{DM}} e : \mathbf{D} a$ w. Then for all $\tilde{v}$ such that $\vdash w \tilde{v} : \mathbf{prop}$, $w \tilde{v}$ implies $e^{\text{HFL}} \tilde{v}$. In particular, in the case of the pure computation monad, $w p$ implies $e^{\text{HFL}} p$ for every predicate p (that represents the condition on the value returned by e) of type $a \rightarrow \mathbf{prop}$.*

4 Differences between the Two Approaches

In the previous section, we have shown a direct correspondence between the Dijkstra monad approach and the HFL(Z) approach for a restricted fragment of the languages supported by the two approaches. In this section, we discuss what differences arise without the restrictions. The major restrictions on the language considered in the previous section are: (1) Restriction on the inner type of the monad. The type a inside the monad $\mathbf{M} a$ is restricted to **unit** or **int**, and not (pure) functional types such as $\mathbf{int} \rightarrow \mathbf{int}$. (2) Restriction to non-nested monads. Nested monadic types like $\mathbf{int} \rightarrow \mathbf{M} (\mathbf{int} \rightarrow \mathbf{M} \mathbf{unit})$ were excluded. (3) Restriction to non-recursive programs. Of the three kinds of restrictions, the first one is not fundamental. The restriction comes from the restriction on the side of HFL(Z). In HFL(Z), the final return type of a function type must be **prop** (i.e., the types of functions or predicates are restricted to the form $\tau_1 \rightarrow \dots \rightarrow \tau_k \rightarrow \mathbf{prop}$, and types like $\mathbf{int} \rightarrow \mathbf{int}$ are disallowed). That restriction is required in HFL(Z) to ensure that the least/greatest fixpoints always exist. A workaround is to wrap the final return type with a monad, such as using $\mathbf{int} \rightarrow \mathbf{M} \mathbf{int}$ instead of $\mathbf{int} \rightarrow \mathbf{int}$. In addition, if we allow only terminating functions by employing a mechanism for checking the termination of functions as in F^* , then there would be no problem in removing the restriction on the HFL(Z) side.

The second and third restrictions are more fundamental, revealing important differences between the two approaches, as discussed below.

4.1 On Nested Monads

To see the difference between the two approaches in the presence of nested monads, let us consider the following program:

```
let nest() = increment(); increment(); return increment
let client() = let f = nest() in f()
```

Here, **increment** is as defined in Section 2. In the DM approach, the function **nest** is given the type $\mathbf{unit} \rightarrow \mathbf{D}^{\text{STATE}} (\mathbf{unit} \rightarrow \mathbf{D}^{\text{STATE}} \mathbf{unit} w_2) w_1$ where $w_1 = \lambda h. \lambda p. p \mathbf{increment} (h + 2)$ and $w_2 = \lambda h. \lambda p. p () (h + 1)$. Note that the information about the behavior of the function is split into w_1 and w_2 , where w_1 and w_2 respectively express the effect when the first and second arguments are given. Note also that the code **increment** is embedded in the former predicate transformer w_1 .

The function `client` is given the type $\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} \text{unit}$ w , where w is calculated as follows (using the β -equality).

$$\begin{aligned} w &= \lambda h. \lambda p. w_1 h (\lambda x. \lambda h'. w_2 h' p) \\ &= \lambda h. \lambda p. (\lambda h. \lambda p. p \text{increment} (h + 2)) h (\lambda x. \lambda h'. w_2 h' p) \\ &= \lambda h. \lambda p. (\lambda x. \lambda h'. w_2 h' p) \text{increment} (h + 2) \\ &= \lambda h. \lambda p. w_2 (h + 2) p = \lambda h. \lambda p. p () (h + 3). \end{aligned}$$

Thus, the code `increment` is actually irrelevant in the computation of the predicate transformer.

The shape of the formula obtained in the HFL(Z) approach is quite different. By applying the transformation in Section 3.3 to the body of the function `nest`, we obtain:

$$\begin{aligned} \varphi_{\text{nest}} &:= \text{bind}^W (\varphi_{\text{inc}}()) \lambda(). (\text{bind}^W (\varphi_{\text{inc}}()) \lambda(). \text{return}^W \varphi_{\text{inc}}) \\ &= \lambda h. \lambda p. p \varphi_{\text{inc}} (h + 2). \end{aligned}$$

where $\varphi_{\text{inc}} = \lambda(). \lambda h. \lambda p. p(), h + 1$, and bind^W and return^W are as given in Section 3.1, Example 2. Thus, the behavior of the function is described in the single predicate φ_{nest} , unlike in the DM approach (where the same information is split into w_1 and w_2). For the client code (whose type does not contain nested monads), however, we obtain the same formula as w :

$$\begin{aligned} \varphi_{\text{client}} &:= \text{bind}^W \varphi_{\text{nest}} \lambda f. f() \\ &= \lambda h. \lambda p. \varphi_{\text{nest}} h (\lambda(x, h'). (\lambda f. f()) x h' p) =_{\beta} \lambda h. \lambda p. p () (h + 3). \end{aligned}$$

Interestingly, the formula $\varphi_{\text{nest}} = \lambda h. \lambda p. p \varphi_{\text{inc}} (h + 2)$ looks quite similar to $w_1 = \lambda h. \lambda p. p \text{increment} (h + 2)$. The only difference is that `increment` in w_1 is code, while φ_{inc} in φ_{nest} is a specification (in the form of a predicate transformer) of the code. Both look even more similar if we replace w_1 with the following predicate transformer w'_1 :

$$w'_1 = \lambda h. \lambda p. \forall f : (\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} \text{unit } w_2). p f (h + 2).^{10}$$

Here, by using the quantification over the code, we avoid embedding the code `increment` and instead provide its specification as a type. Thus, w'_1 contains the same amount of information as φ_{nest} . A difference between the two approaches still remains, however. Even with this modification, f and its type are actually irrelevant in the computation of the predicate transformer for `client`, just as `increment` has been ignored in the computation of w . Information about the behavior of `nest` for the second argument is duplicated in the DM approach, no matter whether we use the type $\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} (\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} \text{unit } w_2) w_1$ or $\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} (\text{unit} \rightarrow \mathsf{D}^{\text{STATE}} \text{unit } w_2) w'_1$. (In the former type, w_1 contains the

¹⁰ Universal quantifications inside predicate transformers are not supported by the fragment defined in Section 3, but are supported by the original type system of Dijkstra monads [15].

code `increment`, whose behavior is already described in w_2 , and in the latter type, w_2 also occurs in w'_1 .) This problem can be annoying for a programmer and can incur unnecessary annotation overheads, especially when the nesting depth of monads is larger, like $\text{int} \rightarrow \mathbb{M} (\text{int} \rightarrow \mathbb{M} (\text{int} \rightarrow \mathbb{M} (\text{int} \rightarrow \mathbb{M} \text{unit})))$. We, therefore, think that the HFL(Z) approach is preferable in this aspect.

4.2 On Recursive Programs

For recursive programs, there are clear differences between the two approaches. In the DM approach, each recursive function must be annotated with an appropriate type (which includes the predicate transformer) by a programmer, just like invariants for loops and recursive functions must be provided in deductive verification. The obtained predicate transformer is not necessarily the weakest one, depending on the precision of the annotations. In the HFL(Z) approach, no annotations are required for recursive functions, and (a predicate corresponding to) the *weakest* predicate transformer can be automatically expressed by using fixpoint operators.

As an example, let us consider the following program (which is admittedly artificial, for the sake of simplicity).

```
let rec add x y = if x ≤ 0 then y else add (x - 1) (y + 1)
let rec zero x = if x ≤ 0 then 0 else zero (x - 1)
let main x = assert (add x (zero x) ≥ x)
```

It first defines recursive functions `add` and `zero`, and then asserts that $\text{add } x (\text{zero } x) \geq x$ holds for every x in the main function. Suppose that we wish to verify that the assertion never fails. In the DM approach, we have to declare the following types:

$$\begin{aligned} \text{add} &: x : \text{int} \rightarrow y : \text{int} \rightarrow \mathbb{D} \text{ int } (\lambda p. \forall r \geq x + y. p \ r) \\ \text{zero} &: x : \text{int} \rightarrow \mathbb{D} \text{ int } (\lambda p. p \ 0). \end{aligned}$$

Then the verification problem boils down to a type checking problem, which is solved by utilizing an underlying SMT solver. One may argue that it is a good programming practice to declare types, but it would be cumbersome to declare the types of *all* the recursive functions, including those of auxiliary functions.

In the HFL(Z) approach, the function `main` is automatically converted to the formula $\varphi_{\text{main}} := \lambda x. \lambda p. P_{\text{zero}} \ x \ (\lambda y. P_{\text{add}} \ x \ y \ (\lambda z. z \geq x))$ where the predicates P_{add} and P_{zero} are defined by:

$$\begin{aligned} P_{\text{add}} \ x \ y \ p &=_{\nu} (x \leq 0 \Rightarrow p \ y) \wedge (x > 0 \Rightarrow P_{\text{add}} \ (x - 1) \ (y + 1) \ p) \\ P_{\text{zero}} \ x \ p &=_{\nu} (x \leq 0 \Rightarrow p \ 0) \wedge (x > 0 \Rightarrow P_{\text{zero}} \ (x - 1) \ p). \end{aligned}$$

Its validity can automatically be proved by an HFL(Z) validity checker [6,8]. Thus, the whole verification process is fully automatic in the HFL(Z) approach.

The property above is a safety property, but one can mix the least and greatest fixpoint equations to express arbitrary regular temporal properties [9,17], and

its validity can be automatically checked (though the backend HFL(Z) validity checker is necessarily incomplete) [8].

Despite the advantages mentioned above, the HFL(Z) approach has also a disadvantage: automated HFL(Z) validity checkers used in the approach are much less efficient and less predictable than SMT solvers used in the DM approach. The inefficiency and unpredictability are inevitable because HFL(Z) validity checking is undecidable in general. In contrast, the DM approach uses the annotations to reduce the validity checking problem to simpler problems that can be discharged by ordinary SMT solvers. In fact, there are programs that the HFL(Z) approach cannot verify in realistic time, but given appropriate annotations, the DM approach can verify; see Section 5.2 for such an example. The DM approach allows modular verification; by using type annotations, the whole verification problem can be decomposed into small pieces of verification conditions.

In summary, the two approaches handle recursion differently, resulting in contrasting advantages and disadvantages:

- In the DM approach, users need to provide a type annotation (including predicate transformers) for each recursive function, whereas in the HFL(Z) approach, there is no such need. The HFL(Z) approach can deal with not only partial/total correctness but also arbitrary regular temporal properties.
- The DM approach requires only an ordinary SMT solver as a backend, which is more efficient and predictable than a higher fixpoint logic solver required in the HFL(Z) approach. The former is thus more suitable for interactive program development.
- The DM approach admits modular verification, and thus it is more scalable to large programs than the HFL(Z) approach, by taking advantage of the annotations given by users.

5 Semi-Automated Program Verification Using ν HFL(Z)

Based on the observations of the similarities and differences between the DM and HFL(Z) approaches, we introduce a semi-automated verification method to take the best of two approaches in this section.

5.1 Overview of Our Approach

Our method to integrate the DM and HFL(Z) approaches is to allow *partial* annotations of recursive functions using (a restricted form of) DM types, and then reduce the verification problem to the refinement type checking problem for ν HFL(Z) [6]. Let us reconsider the example in Section 4.2. Recall that type annotations have to be provided for both **add** and **zero** in the DM approach, and none are provided in the HFL(Z) approach. In our new approach, we allow a user to provide type annotations partially; for example, a user may choose not to annotate **zero**, but to annotate **add** as follows, asserting that **add** x y returns $x + y$:

$$\mathbf{add} : x : \mathbf{int} \rightarrow y : \mathbf{int} \rightarrow \mathbf{H}^{\mathbf{pure}} \mathbf{int} \mathbf{true} \lambda r. r \geq x + y.$$

Here, the type constructor $\mathbb{H}^{\text{pure}}$ is a kind of Hoare monad [13], defined by:

$$\mathbb{H}^{\text{pure}} a \text{ pre post} := D a (\lambda p. \text{pre} \wedge \forall r. \text{post } r \Rightarrow p).$$

(This encoding follows [15], Section 3.1.)

Given such a partially annotated program, we can convert it to a $\text{HFL}(\mathbb{Z})$ formula as in Section 3.3, and, additionally, translate the type annotations for recursive functions to refinement types for the corresponding $\text{HFL}(\mathbb{Z})$ predicates [6,7]. In the above case, the type annotation for `add` can be converted to the following type annotation for the corresponding $\text{HFL}(\mathbb{Z})$ predicate:

$$P_{\text{Add}} : (x : \text{int} \rightarrow y : \text{int} \rightarrow (r : \text{int} \rightarrow \bullet[r \geq x + y]) \rightarrow \bullet[\text{true}]).$$

Here, $\bullet[\theta]$ denotes the type of a proposition that holds whenever θ is true. Thus, the type above means that $P_{\text{Add}} x y p$ is true if $p(x+y)$ is true. In general, the type $\mathbb{H}^{\text{pure}} a \text{ pre post}$ for a term in the source language can be automatically converted to $(r : a \rightarrow \bullet[\text{post } r]) \rightarrow \bullet[\text{pre}]$ for the corresponding $\text{HFL}(\mathbb{Z})$ formula. We can then decompose the verification problem to type checking problems for $\nu\text{HFL}(\mathbb{Z})$ (which is a fragment of $\text{HFL}(\mathbb{Z})$ with only greatest fixpoint operators) [6,7].

In general, given (i) a main formula φ , (ii) a set $D_1 \cup D_2$ of fixpoint equations on $\nu\text{HFL}(\mathbb{Z})$ predicates, and (iii) type environments Δ_1 for D_1 (which are automatically derived from type annotations), we can reduce the validity of φ to the following type checking problems, thanks to the soundness of the refinement type system for $\nu\text{HFL}(\mathbb{Z})$ [6,7].

- Are the type annotations valid, i.e., does there exist a type environment Δ_2 for D_2 , such that $\Delta_1, \Delta_2 \vdash^{\text{RT}} \varphi_P : \rho$ for each type binding $P : \rho$ in Δ_1, Δ_2 ?
- Is the main formula valid, i.e., does there exist a type environment Δ_2 for D_2 , such that $\Delta_1, \Delta_2 \vdash^{\text{RT}} \varphi : \bullet[\text{true}]$ and $\Delta_1, \Delta_2 \vdash^{\text{RT}} \varphi_P : \rho$ for each type binding $P : \rho$ in Δ_2 ?

Here, $\Delta \vdash^{\text{RT}} \varphi : \rho$ is the type judgment relation as defined in [6] (whose details are not important here) and φ_P is the body of P . Thus, partial type annotations enable modular verification, which is expected to be more scalable and predictable than the original $\text{HFL}(\mathbb{Z})$ approach. In the above decomposition of the verification problem, the weakest predicate transformers are automatically computed and used for unannotated functions, thanks to the expressive power of $\text{HFL}(\mathbb{Z})$. (In contrast, the implementation of DM in F^* does seem to allow some type annotations to be omitted, but in that case, very imprecise predicate transformers would be used for unannotated recursive functions.)

5.2 Experiments

We extended the existing type-based $\nu\text{HFL}(\mathbb{Z})$ validity checker, `RETHFL` [6] to allow partial type annotations, and conducted experiments to confirm the effectiveness of our approach.¹¹ Currently, the type annotations need to be provided for $\nu\text{HFL}(\mathbb{Z})$ formulas rather than for source programs. As discussed in

¹¹ An artifact containing our tool and a script to reproduce the experiment is publicly available at <https://doi.org/10.5281/zenodo.14614021>.

Table 2. Comparison between the original fully automated method by Katsura et al. [6] and our semi-automated method (timeout = 300 seconds)

Program	Method	Time [sec]
array_init (320 B)	Katsura et al.'s	24.24
	Ours (Init annotated, 154 B)	0.02 / 0.01
kmp (1.10 KB)	Katsura et al.'s	Timeout
	Ours (Loop annotated, 321 B)	0.22 / 3.36
	Ours (LoopShift annotated, 425 B)	0.51 / 0.29
memoization (1.71 KB)	Katsura et al.'s	Timeout
	Ours (Fib annotated, 149 B)	0.01 / 0.02
merge (2.18 KB)	Katsura et al.'s	32.93
	Ours (Merge annotated, 215 B)	0.13 / 0.18
soli (4.00 KB)	Katsura et al.'s	27.60
	Ours (Solve annotated, 159 B)	0.21 / 0.07
up3 (329 B)	Katsura et al.'s	Timeout
	Ours (LoopA annotated, 127 B)	0.02 / 0.02
	Ours (LoopB annotated, 68 B)	0.01 / 0.11

Section 5.1, it is easy to automatically convert the Hoare-monad-style type annotations for source programs (which may be considered a restricted form of DM-style type annotations) to type annotations for $\nu\text{HFL}(\mathcal{Z})$ formulas; it is left for future work to implement such a converter to make annotations easier.

In the experiment we used a Linux server equipped with Intel Xeon Gold 6242 Processor @ 2.80 GHz, with 16 cores, and 64 GB of RAM. For each (sub-)problem, we ran each tool five times and calculated the average execution time. Since the performance of the validity checker depends strongly on the performance of the underlying Constrained Horn Clause (CHC) solver, we used both Z3/SPACER [10] and HoIce [3] as underlying CHC solvers and show only the best of their results for each problem; in other words, the reported verification times indicate those in case the backend solvers Z3/SPACER [10] and HoIce [3] are called in parallel.

As the verification target of experiments, we gathered, from a benchmark for higher-order functional program verification¹², programs for which the current RETHFL is slow or timeouts.¹³ The LOCs of the target programs varies from 10 to 153. For each program, we provided a refinement type as an annotation for one of the recursive predicates (which corresponds to one of the recursive functions in the source program). See the longer version [19] for more details about the benchmark programs and an example of type annotations.

The experimental results are shown in Table 2. In the table, we show the sizes of programs and annotations in terms of bytes for a fair comparison between the sizes of programs and annotations (as it is possible to write a type annotation

¹² <https://github.com/hopv/benchmarks>

¹³ There are a few other programs in the benchmark that RETHFL cannot verify, but they are due to a certain theoretical limitation of the underlying refinement type system of RETHFL, which cannot be remedied by type annotations.

in one long line). The first row of each program shows the result of the original version of `ReTHFL`. As it is a fully automated checker, no annotations were provided. The rest of the rows show the results of our method. The “Method” column shows the name of the function annotated and the annotation size. In the rest of the columns, “X / Y” means that X is the result of the sub-problem (a) that checks the correctness of the annotation and Y is the result of the sub-problem (b) that checks the validity of the whole formula.

For all runs of each sub-problem, our tool successfully checked the validity and terminated within a few seconds. This confirms the effectiveness of our approach and the behavior of the tool is much more predictable than the fully-automated verification using the original `ReTHFL`. We think the size of annotations is acceptable; actually, the annotation size would become smaller, if we allow a user to provide annotations for source programs rather than for `HFL(Z)` formulas.

The results also support the advantage of our method over the DM approach in reducing the annotation burdens. While the DM approach requires an annotation for every recursive function, our method succeeded given only one annotated function for each program.

6 Related Work

The Dijkstra Monad Approach and the `HFL(Z)` Approach. The framework of Dijkstra monads was first introduced by Swamy et al. [15] as a general method for the verification of higher-order programs with monadic effects. To define a Dijkstra monad that is consistent with the semantics of the effect, Ahman et al. [1] showed that CPS transformation of the monadic model of the effect yields such a correct monad. This result is closely related to our observation of the similarity with the `HFL(Z)` approach. As we mentioned in Section 3.4, Lemma 1, which is part of the proof of the correspondence theorem, generalizes Theorem 6 of Ahman et al. to be parameterized by effect parameters. A successive study by Maillard et al. [12] reinterpreted and extended the framework by considering a Dijkstra monad as two monads—a computational monad and a specification monad—related by a monad morphism.

The `HFL(Z)` approach was proposed by Kobayashi et al. [9,17], which supports the verification of temporal properties of higher-order programs with general recursion. Watanabe et al. [17] extended the approach to the verification of arbitrary regular linear/branching-time properties expressible in alternating parity tree automata. Following those studies, fully automated `HFL(Z)` validity checkers have been developed [8,5,6,7], which, in combination with the above-mentioned reductions [9,17], yielded fully automated verification tools for higher-order functional programs. As discussed already, while the fully automated approach is attractive for verification of small programs, it suffers from (non-)scalability and unpredictability issues for large programs.

Most of the previous studies on Dijkstra monads have considered only partial or total correctness, while the `HFL(Z)` approach supports a wide range of temporal properties of possibly divergent programs. An exception is the work of Silver

and Zdancewic [14], which can deal with termination-sensitive properties of possibly divergent programs with monadic effects. The behavior of such programs is expressed using interaction trees [18], a coinductive data structure with effects as internal nodes and resulting values as leaves. The idea of expressing computations as possibly infinite trees is shared with the work of Watanabe et al. [17] in the HFL(Z) approach, where the behavior of a possibly non-terminating program is modeled by a possibly infinite tree of (abstract) events. It is left for future work to extend our result in Section 3 to establish a precise relationship between those studies.

Predicate Transformers for Higher-Order Programs. Kura [11] considered a general categorical framework for a higher-order language with effects and recursion, and showed that the (semantically defined) weakest precondition transformer of a program can be obtained by the CPS transformation. He has shown that a fragment of the HFL(Z) approach [9] can be understood as an instance of his result. He did not show, however, a direct correspondence between the DM and HFL(Z) approaches. Another related approach to higher-order program verification based on predicate transformers is Charguéraud’s work [4]. He showed a construction of a characteristic formula $\llbracket t \rrbracket$ for a term t , such that $\llbracket t \rrbracket P$ is valid if and only if t satisfies the postcondition P . Thus, $\llbracket t \rrbracket$ can be viewed as a predicate transformer. Charguéraud’s approach is motivated by interactive verification using a proof assistant, while the HFL(Z) approach is motivated by automated verification. This difference in the motivations seems to have led to the difference in the choice of logics. Charguéraud used a more expressive higher-order logic (like that of Coq) than HFL(Z) to express characteristic formulae so that the characteristic formula $\llbracket t \rrbracket$ has almost the same shape as the original term t . In contrast, HFL(Z) is designed to be a minimal higher-order logic for ease of automation.

7 Conclusion

This paper studied the relationship between two approaches to higher-order program verification: the Dijkstra monad approach and the HFL(Z) approach. We showed the correspondence between them in a restricted setting, and discussed the differences between the two approaches outside the restriction. Although the two approaches are closely related in the restricted fragment, the full systems have some different strengths. Based on this observation, we proposed a semi-automated verification method and experimentally confirmed that our method can verify programs that cannot be verified by the existing HFL(Z)-based automated tool, with fewer annotations than in the DM approach. A full integration between the two approaches is left for future work.

Acknowledgment. We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Numbers Grant JP20H05703 and JP24K20731.

References

1. Ahman, D., Hritcu, C., Maillard, K., Martínez, G., Plotkin, G.D., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: Castagna, G., Gordon, A.D. (eds.) *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. pp. 515–529. ACM (2017), <https://doi.org/10.1145/3009837.3009878>
2. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, K.R.M., Lorch, J.R., Maillard, K., Pan, J., Parno, B., Protzenko, J., Ramananandro, T., Rane, A., Rastogi, A., Swamy, N., Thompson, L., Wang, P., Béguelin, S.Z., Zinzindohoue, J.K.: Everest: Towards a verified, drop-in replacement of HTTPS. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA. LIPIcs*, vol. 71, pp. 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017), <https://doi.org/10.4230/LIPIcs.SNAPL.2017.1>
3. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. *J. Autom. Reason.* **64**(7), 1393–1418 (2020), <https://doi.org/10.1007/s10817-020-09571-y>
4. Charguéraud, A.: Program verification through characteristic formulae. In: Hudak, P., Weirich, S. (eds.) *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. pp. 321–332. ACM (2010), <https://doi.org/10.1145/1863543.1863590>
5. Iwayama, N., Kobayashi, N., Suzuki, R., Tsukada, T.: Predicate abstraction and CEGAR for νHFL_z validity checking. In: Pichardie, D., Sighireanu, M. (eds.) *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings. Lecture Notes in Computer Science*, vol. 12389, pp. 134–155. Springer (2020), https://doi.org/10.1007/978-3-030-65474-0_7
6. Katsura, H., Iwayama, N., Kobayashi, N., Tsukada, T.: A new refinement type system for automated νHFL_z validity checking. In: d. S. Oliveira, B.C. (ed.) *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. Lecture Notes in Computer Science*, vol. 12470, pp. 86–104. Springer (2020), https://doi.org/10.1007/978-3-030-64437-6_5
7. Katsura, H., Kobayashi, N., Sato, R.: Higher-order property-directed reachability. *Proc. ACM Program. Lang.* **7**(ICFP) (aug 2023), <https://doi.org/10.1145/3607831>
8. Kobayashi, N., Tanahashi, K., Sato, R., Tsukada, T.: HFL(Z) validity checking for automated program verification. *Proc. ACM Program. Lang.* **7**(POPL), 154–184 (2023), <https://doi.org/10.1145/3571199>
9. Kobayashi, N., Tsukada, T., Watanabe, K.: Higher-order program verification via HFL model checking. In: Ahmed, A. (ed.) *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science*, vol. 10801, pp. 711–738. Springer (2018), https://doi.org/10.1007/978-3-319-89884-1_25
10. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) *Computer Aided Verification - 25th International Conference, CAV 2013*,

- Saint Petersburg, Russia, July 13–19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 846–862. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59
11. Kura, S.: Higher-order weakest precondition transformers via a CPS transformation (2023), <https://doi.org/10.48550/arXiv.2301.09997>
 12. Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. *Proc. ACM Program. Lang.* **3**(ICFP), 104:1–104:29 (2019), <https://doi.org/10.1145/3341708>
 13. Nanevski, A., Morrisett, J.G., Birkedal, L.: Hoare type theory, polymorphism and separation. *J. Funct. Program.* **18**(5–6), 865–911 (2008), <https://doi.org/10.1017/S0956796808006953>
 14. Silver, L., Zdancewic, S.: Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* **5**(POPL), 1–28 (2021), <https://doi.org/10.1145/3434307>
 15. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying higher-order programs with the Dijkstra monad. In: Boehm, H., Flanagan, C. (eds.) *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, Seattle, WA, USA, June 16–19, 2013. pp. 387–398. ACM (2013), <https://doi.org/10.1145/2491956.2491978>
 16. Viswanathan, M., Viswanathan, R.: A higher order modal fixed point logic. In: *CONCUR. LNCS*, vol. 3170, pp. 512–528. Springer (2004)
 17. Watanabe, K., Tsukada, T., Oshikawa, H., Kobayashi, N.: Reduction from branching-time property verification of higher-order programs to HFL validity checking. In: Hermenegildo, M.V., Igarashi, A. (eds.) *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019*, Cascais, Portugal, January 14–15, 2019. pp. 22–34. ACM (2019), <https://doi.org/10.1145/3294032.3294077>
 18. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* **4**(POPL), 51:1–51:32 (2020), <https://doi.org/10.1145/3371119>
 19. Yamada, R., Kobayashi, N., Sakayori, K., Sato, R.: On the relationship between dijkstra monads and higher-order fixpoint logic. A longer version, available from the second author's webpage (2025)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Author Index

A

Accattoli, Beniamino I-1
Acclavio, Matteo I-23
Ambal, Guillaume I-56

B

Bernardi, Giovanni I-88
Birkedal, Lars II-286
Blazy, Sandrine II-91, II-119
Boillot, Jérôme I-117
Bond, Joe I-144
Borthelle, Peio I-172, I-200
Bucev, Mario II-154

C

Castellani, Ilaria I-88
Chan, Jonathan I-236
Chassot, Samuel II-154
Cordeiro, Lucas C. I-206

D

D'Ousualdo, Emanuele II-314
Daggitt, Matthew L. I-206
Dardha, Ornella II-125
David, Cristina I-144
Demange, Delphine II-91, II-119

E

Ehrhard, Thomas I-292
Eremondi, Joseph I-264

F

Faggian, Claudia I-292
Fan, Weijie I-322
Feng, Xinyu I-322, I-379
Feret, Jérôme I-117
Filliâtre, Jean-Christophe II-175
Fowler, Simon II-125
Fromherz, Aymeric II-31
Frumin, Dan II-286

G

Germane, Kimball II-374
Girard-Satabin, Julien I-206
Giunti, Marco I-349
Goharshady, Amir K. II-1
Goutagny, Pierre II-31
Gutierrez, Claudio I-408

H

Hirschowitz, Tom I-172, I-200
Hitarth, S. II-1

I

Isac, Omri I-206

J

Jaber, Guilhem I-172, I-200
Jiang, Hanru I-322
Johnson, Taylor T. I-206
Jongmans, Sung-Shik II-62

K

Kammar, Ohad I-264
Katz, Guy I-206
Kobayashi, Naoki II-402
Komendantskaya, Ekaterina I-206
Kunčák, Viktor II-154

L

La Spina, Roméo II-91, II-119
Laforgue, Paul I-88
Lahav, Ori I-56
Le Brun, Matthew Alan II-125
Lemesle, Augustin I-206
Liang, Hongjin I-322, I-379
Lin, Rongen I-379

M

Manara, Giulia I-23
Manino, Edoardo I-206
Milovančević, Dragana II-154

Monat, Raphaël [II-31](#)
 Montesi, Fabrizio [I-23](#)

N

Nardino, Emma [II-286](#)
 Near, Joseph P. [II-258](#)
 Nguyen, Minh [I-144](#)
 Novozhilov, Sergei [II-1](#)

O

Oliveira, Bruno C. d. S. [I-408](#), [II-347](#)
 Orchard, Dominic [I-144](#)

P

Pagani, Michele [I-292](#)
 Paskevich, Andrei [II-175](#)
 Patault, Paul [II-175](#)
 Perera, Roly [I-144](#)
 Piedeleu, Robin [II-202](#)

R

Raad, Azalea [I-56](#)
 Rogalewicz, Adam [II-230](#)

S

Sakayori, Ken [II-402](#)
 Sato, Ryosuke [II-402](#)
 Sextl, Florian [II-230](#)
 Silva, Alexandra [II-202](#)
 Šinkarovs, Artjoms [I-206](#)

Skalka, Christian [II-258](#)
 Stefanescu, Léo [I-88](#)
 Stepanenko, Sergei [II-286](#)
 Stutz, Felix [II-314](#)
 Sun, Yaozhu [II-347](#)

T

Tanter, Éric [I-408](#)
 Timany, Amin [II-286](#)
 Toro, Matías [I-408](#)
 Torres-Ruiz, Mateo [II-202](#)

V

Vojnar, Tomáš [II-230](#)

W

Weirich, Stephanie [I-236](#)
 Whiting, Tim [II-374](#)
 Wojnarowski, Marcin [II-154](#)
 Wu, Haoze [I-206](#)

Y

Yamada, Risa [II-402](#)
 Ye, Wenjia [I-408](#)
 Yoshida, Nobuko [I-349](#)

Z

Zakowski, Yannick [I-172](#), [I-200](#)
 Zanasi, Fabio [II-202](#)
 Zuleger, Florian [II-230](#)