LNCS 14577

ARCoSS

Stephanie Weirich (Ed.)

Programming Languages and Systems

33rd European Symposium on Programming, ESOP 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part II

2 Part II

ETAPS
EUROPEAN JOINT CONFERENCES ON
THEORY & PRACTICE OF SOFTWARE

Springer

OPEN ACCESS

Stephanie Weirich
Editor

Programming Languages and Systems

33rd European Symposium on Programming, ESOP 2024
Held as Part of the European Joint Conferences
on Theory and Practice of Software, ETAPS 2024
Luxembourg City, Luxembourg, April 6–11, 2024
Proceedings, Part II

Springer

Editor
Stephanie Weirich
University of Pennsylvania
Philadelphia, PA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-031-57266-1 ISBN 978-3-031-57267-8 (eBook)
https://doi.org/10.1007/978-3-031-57267-8

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

Paper in this product is recyclable.

ETAPS Foreword

Welcome to the 27th ETAPS! ETAPS 2024 took place in Luxembourg City, the beautiful capital of Luxembourg.

ETAPS 2024 is the 27th instance of the European 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. Organising 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 2024 received 352 submissions in total, 117 of which were accepted, yielding an overall acceptance rate of 33%. 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 2024 featured the unifying invited speakers Sandrine Blazy (University of Rennes, France) and Lars Birkedal (Aarhus University, Denmark), and the invited speakers Ruzica Piskac (Yale University, USA) for TACAS and Jérôme Leroux (Laboratoire Bordelais de Recherche en Informatique, France) for FoSSaCS. Invited tutorials were provided by Tamar Sharon (Radboud University, the Netherlands) on computer ethics and David Monniaux (Verimag, France) on abstract interpretation.

As part of the programme we had the first ETAPS industry day. The goal of this day was to bring industrial practitioners into the heart of the research community and to catalyze the interaction between industry and academia. The day was organized by Nikolai Kosmatov (Thales Research and Technology, France) and Andrzej Wąsowski (IT University of Copenhagen, Denmark).

ETAPS 2024 was organized by the SnT - Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. The University of Luxembourg was founded in 2003. The university is one of the best and most international young universities with 6,000 students from 130 countries and 1,500 academics from all over the globe. The local organisation team consisted of Peter Y.A. Ryan (general chair), Peter B. Roenne (organisation chair), Maxime Cordy and Renzo Gaston Degiovanni (workshop chairs), Magali Martin and Isana Nascimento (event manager), Marjan Skrobot (publicity chair), and Afonso Arriaga (local proceedings chair). This team also organised the online edition of ETAPS 2021, and now we are happy that they agreed to also organise a physical edition of ETAPS.

ETAPS 2024 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), Arnd Hartmanns (Twente), Caterina Urban (Inria), Jan Křetínský (Munich), Elizabeth Polgreen (Edinburgh), and Lenore Zuck (Chicago).

Other members of the steering committee are: Maurice ter Beek (Pisa), Dirk Beyer (Munich), Artur Boronat (Leicester), Luís Caires (Lisboa), Ana Cavalcanti (York), Ferruccio Damiani (Torino), Bernd Finkbeiner (Saarland), Gordon Fraser (Passau), Arie Gurfinkel (Waterloo), Reiner Hähnle (Darmstadt), Reiko Heckel (Leicester), Marijn Heule (Pittsburgh), Joost-Pieter Katoen (Aachen and Twente), Delia Kesner (Paris), Naoki Kobayashi (Tokyo), Fabrice Kordon (Paris), Laura Kovács (Vienna), Mark Lawford (Hamilton), Tiziana Margaria (Limerick), Claudio Menghi (Hamilton and Bergamo), Andrzej Murawski (Oxford), Laure Petrucci (Paris), Peter Y.A. Ryan (Luxembourg), Don Sannella (Edinburgh), Viktor Vafeiadis (Kaiserslautern), Stephanie Weirich (Pennsylvania), Anton Wijs (Eindhoven), and James Worrell (Oxford).

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 2024 was also generously supported by a RESCOM grant from the Luxembourg National Research Foundation (project 18015543). I hope you all enjoyed ETAPS 2024.

Finally, a big thanks to both Peters, Magali and Isana and their local organization team for all their enormous efforts to make ETAPS a fantastic event.

April 2024

Marieke Huisman
ETAPS SC Chair
ETAPS e.V. President

Preface

These proceedings volumes contain papers that were presented at the 33rd European Symposium on Programming (ESOP 2024), held during April 6–11 in Luxembourg City, Luxembourg, along with associated artifact reports. ESOP is part of the European 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 include 25 research papers, one "fresh perspective" and four "artifact reports". The latter two paper categories are new to ESOP. In addition to standard research papers, the ESOP 2024 call-for-papers included the new submission categories: "fresh perspectives" that provide new insights in a particularly elegant way and "experience reports" that describe tools and systems used in practice. Furthermore, authors of accepted papers were allowed to submit short "artifact reports", to appear together with their research papers, that describe associated software, tools, data sets, or machine checked proofs to substantiate the claims made in their papers.

The papers in this volume were selected from 66 papers submitted in the research paper category and 6 papers submitted in the "fresh perspectives" category. There were no submissions for "experience reports". While papers in these new categories had strict formatting requirements, ESOP 2024 allowed research papers to be submitted in any format, of any length, under the advisement that the final paper should be formatted to fit this volume. Fourteen submissions took advantage of this flexibility.

Each submitted paper received at least three reviews by the members of the ESOP program committee. The median PC member was assigned eight papers to review over the seven week review period. In some cases, PC members solicited additional reviews to aid in the decision making process. In total, 39 external reviewers added their insight to the paper selection process. ESOP employed full double-blind review and author identities were only revealed to reviewers on paper acceptance. Authors were also given a chance to respond to their reviews, before the program was selected through a two week online, asynchronous PC meeting, facilitated by the EasyChair system. The program chair had no conflicts with any submitted paper.

ESOP 2024 also employed an artifact evaluation process. Nineteen of the 26 accepted papers elected to make their artifacts available on the archive sites Zenodo and figshare. The committee awarded the badge "Functional" to five of these and the badges "Functional and reusable" to the remaining fourteen. Four accepted papers in this volume are accompanied by artifact reports. These reports were all accepted following a light review by both the program committee and the ESOP/FASE/FoSSaCS joint artifact evaluation committee.

Indeed, 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. Tobias Kappé 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 and the local proceedings chair Afonso Delerue Arriaga, and webmaster Jan Kofroň for their assistance in fitting ESOP together with the entire ETAPS meeting. Finally, thanks are due to the members of the ESOP steering committee. In particular, Luis Caires, as chair of the SC, was a constant source of support, encouragement, information and guidance.

April 2024

Stephanie Weirich
ESOP PC Chair

Organization

ESOP Steering Committee

Luis Caires	Universidade Nova de Lisboa, Portugal
Brigitte Pientka	McGill University, Canada
Ilya Sergey	Yale-NUS College and National University of Singapore, Singapore
Thomas Wies	New York University, USA
Nobuko Yoshida	Imperial College London, UK
Victor Vafaiedis	Max Planck Institute for Software and Systems, Germany

Program Chair

Stephanie Weirich	University of Pennsylvania, USA

Program Committee

Ana Bove	Chalmers University of Technology, Sweden
Loris D'Antoni	University of Wisconsin-Madison, USA
Ugo Dal Lago	Università di Bologna and Inria Sophia Antipolis, Italy
Ornela Dardha	University of Glasgow, UK
Mike Dodds	University of York, UK
Sophia Drossopoulou	Imperial College London, UK
Robby Findler	Northwestern University, USA
Amir Goharshady	Hong Kong University of Science and Technology, China
Andrew D. Gordon	Microsoft Research and University of Edinburgh, UK
Alexey Gotsman	IMDEA Software Institute, Spain
Limin Jia	Carnegie Mellon University, USA
Josh Ko	Institute of Information Science, Academia Sinica, Taiwan
András Kovács	Eötvös Loránd University, Hungary
Kazutaka Matsuda	Tohoku University, Japan
Anders Miltner	Simon Fraser University, Canada
Santosh Nagarakatte	Rutgers University, USA
Dominic Orchard	University of Kent, UK
Frank Pfenning	Carnegie Mellon University, USA
Clément Pit-Claudel	EPFL, Switzerland
François Pottier	Inria, France
Matija Pretnar	University of Ljubljana, Slovenia
Azalea Raad	Imperial College London, UK

James Riely	DePaul University, USA
Tom Schrijvers	Katholieke Universiteit Leuven, Belgium
Peter Sewell	University of Cambridge, UK
Takeshi Tsukada	Chiba University, Japan
Benoît Valiron	LMF - CentraleSupelec, Univ. Paris Saclay, France
Dimitrios Vytiniotis	Google DeepMind, UK
Elena Zucca	DIBRIS - University of Genova, Italy

ESOP/FASE/FoSSaCS Joint Artifact Evaluation Committee

AEC Co-chairs

Tobias Kappé	Open Universiteit and ILLC, University of Amsterdam, The Netherlands
Ryosuke Sato	University of Tokyo, Japan
Stefan Winter	LMU Munich, Germany

AEC Members

Arwa Hameed Alsubhi	University of Glasgow, UK
Levente Bajczi	Budapest University of Technology and Economics, Hungary
James Baxter	University of York, UK
Matthew Alan Le Brun	University of Glasgow, UK
Laura Bussi	University of Pisa, Italy
Gustavo Carvalho	Universidade Federal de Pernambuco, Brazil
Chanhee Cho	Carnegie Mellon University, USA
Ryan Doenges	Northeastern University, USA
Zainab Fatmi	University of Oxford, UK
Luke Geeson	University College London, UK
Hans-Dieter Hiep	Leiden University, Belgium
Philipp Joram	Tallinn University of Technology, Estonia
Ulf Kargén	Linköping University, Sweden
Hiroyuki Katsura	University of Tokyo, Japan
Calvin Santiago Lee	Reykjavík University, Iceland
Livia Lestingi	Politecnico di Milano, Italy
Nuno Macedo	University of Porto and INESC TEC, Portugal
Kristóf Marussy	Budapest University of Technology and Economics, Hungary
Ivan Nikitin	University of Glasgow, UK
Hugo Pacheco	University of Porto, Portugal
Lucas Sakizloglou	Brandenburgische Technische Universität Cottbus-Senftenberg, Germany
Michael Schröder	TU Wien, Austria
Michael Schwarz	TU Munich, Germany
Wenjia Ye	University of Hong Kong, China

Additional Reviewers

Contents – Part II

Contents – Part I

Quantum Programming/Domain-Specific Languages

Circuit Width Estimation via Effect Typing and Linear Dependency

Andrea Colledan[1,2](✉) and Ugo Dal Lago[1,2]

[1] University of Bologna, Bologna, Italy
[2] INRIA Sophia Antipolis, Valbonne, France
{andrea.colledan,ugo.dallago}@unibo.it

Abstract. Circuit description languages are a class of quantum programming languages in which programs are classical and produce a *description* of a quantum computation, in the form of a *quantum circuit*. Since these programs can leverage all the expressive power of high-level classical languages, circuit description languages have been successfully used to describe complex and practical quantum algorithms, whose circuits, however, may involve many more qubits and gate applications than current quantum architectures can actually muster. In this paper, we present Proto-Quipper-R, a circuit description language endowed with a linear dependent type-and-effect system capable of deriving parametric upper bounds on the width of the circuits produced by a program. We prove both the standard type safety results and that the resulting resource analysis is correct with respect to a big-step operational semantics. We also show that our approach is expressive enough to verify realistic quantum algorithms.

Keywords: Effect Typing · Lambda Calculus · Quantum Computing · Quipper

1 Introduction

With the promise of providing efficient algorithmic solutions to many problems [11,27,31], some of which are traditionally believed to be intractable [54], quantum computing is the subject of intense investigation by various research communities within computer science, not least that of programming language theory [24,43,51]. Various proposals for idioms capable of tapping into this new computing paradigm have appeared in the literature since the late 1990s. Some of these approaches turn out to be fundamentally new [1,49,52], while many others are strongly inspired by classical languages and traditional programming paradigms [44,48,53,63].

One of the major obstacles to the practical adoption of quantum algorithmic solutions is the fact that despite huge efforts by scientists and engineers alike, it seems that reliable quantum hardware, contrary to classical one, does not scale too easily: although quantum architectures with up to a couple hundred qubits have recently seen the light [9,10,38], it is not yet clear whether the so-called quantum advantage [45] is a concrete possibility, given the tremendous challenges posed by the quantum decoherence problem [50].

This entails that software which makes use of quantum hardware must be designed with great care: whenever part of a computation has to be run on quantum hardware, the amount of resources it needs, and in particular the amount of qubits it uses, should be kept to a minimum. More generally, a fine control over the low-level aspects of the computation, something that we willingly abstract from when dealing with most forms of classical computation, should be exposed to the programmer in the quantum case. This, in turn, has led to the development and adoption of many domain-specific programming languages and libraries in which the programmer *explicitly* manipulates qubits and quantum circuits, while still making use of all the features of a high-level classical programming language. This is the case of the `Qiskit` and `Cirq` libraries [17], but also of the `Quipper` language [25,26].

At the fundamental level, `Quipper` is a circuit description language embedded in `Haskell`. Because of this, `Quipper` inherits all the expressiveness of the high level, higher-order functional programming language that is its host, but for the same reason it also lacks a formal semantics. Nonetheless, over the past few years, a number of calculi, collectively known as the Proto-Quipper language family, have been developed to formalize interesting fragments and extensions of `Quipper` in a type-safe manner [46,48]. Extensions include, among others, dynamic lifting [8,21,35] and dependent types [20,22], but resource analysis is still a rather unexplored research direction in the Proto-Quipper community [56].

The goal of this work is to show that type systems indeed enable the possibility of reasoning about the size of the circuits produced by a Proto-Quipper program. Specifically, we show how linear dependent types in the style of Gaboardi and Dal Lago [12,14,15,23] can be adapted to Proto-Quipper, allowing to derive upper bounds on circuit widths that are parametric on the number of input wires to the circuit, be they classical or quantum. This enables a form of static analysis of the resource consumption of circuit families and, consequently, of the quantum algorithms described in the language. Technically, a key ingredient of this analysis, besides linear dependency, is a novel form of effect typing in which the quantitative information coming from linear dependency informs the effect system and allows it to keep circuit widths under control.

The rest of the paper is organized as follows. Section 2 informally explores the problem of estimating the width of circuits produced by `Quipper`, while also introducing the language. Section 3 provides a more formal definition of the Proto-Quipper language. In particular, it gives an overview of the system of simple types due to Rios and Selinger [46], which however is not meant to reason about the size of circuits. We then move on to the most important technical contribution of this work, namely the linear dependent and effectful type system, which is introduced in Section 4 and proven to guarantee both type safety and a form of total correctness in Section 5. Section 6 is dedicated to an example of a practical application of our type and effect system, that is, a program that builds the Quantum Fourier Transform (QFT) circuit [11,39] and which is verified to do so without any ancillary qubits.

To conclude this introduction, we wish to emphasize that while it is true that quantum computing can be a difficult and intimidating subject, the class of languages analyzed in this work focuses on circuit *construction*, which is an entirely classical process, paying little to no concern to the actual quantum semantics of circuit *execution*. Because of this, and due to space constraints, we refrain from providing a general introduction to quantum computing in this paper. Instead, we refer the interested reader to the excellent works of Nielsen and Chuang [39], Yanofsky and Mannucci [60], and Mingsheng [61].

2 An Overview on Circuit Width Estimation

`Quipper` allows programmers to describe quantum circuits in a high-level and elegant way, using both gate-by-gate and circuit transformation approaches. `Quipper` also supports hierarchical and parametric circuits, thus promoting a view in which circuits become first-class citizens. `Quipper` has been shown to be scalable, in the sense that it has been effectively used to describe complex quantum algorithms that easily translate to circuits involving trillions of gates applied to millions of qubits. The language allows the programmer to optimize the circuit, e.g. by using ancilla qubits for the sake of reducing the circuit depth, or recycling qubits that are no longer needed.

One feature that `Quipper` lacks is a methodology for *statically* proving that important parameters — such as the the width — of the underlying circuit are below certain limits, which of course would need to be parametric on the input size of the circuit. If this kind of analysis were available, then it would be possible to derive bounds on the number of qubits needed to solve any instance of a problem, and ultimately to know in advance how big of an instance can be *possibly* solved given a fixed amount of qubits.

In order to illustrate the kind of scenario we are reasoning about, this section offers some simple examples of `Quipper` programs, showing in what sense we can think of capturing the quantitative information that we are interested in through types and effect systems and linear dependency. We proceed at a very high level for now, without any ambition of formality.

Let us start with the example of Figure 1. The `Quipper` function on the left builds the structure on the right, which we call a *quantum circuit*. For the purposes of this work, it suffices to say that horizontal lines represent qubits, while other symbols represent elementary operations applied to them, e.g. initializations, gate applications, and so on. Time flows from left to right. The specific circuit in Figure 1 consists in an (admittedly contrived) implementation of the quantum not operation. The `dumbNot` function implements negation using a controlled not gate and an ancillary qubit `a`, which is initialized and discarded within the body of the function. This qubit does not appear in the interface of the circuit, but it clearly adds to its overall width, which is 2.

Consider now the higher-order function in Figure 2. This function takes as input a circuit building function `f`, an integer `n` and describes the circuit obtained by applying `f`'s circuit `n` times to the input qubit `q`. It is easy to see that the

```
dumbNot :: Qubit -> Circ Qubit
dumbNot q = do
 a <- qinit True
 (q,a) <- controlled_not q a
 qdiscard a
 return q
```

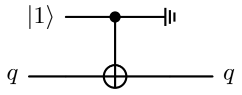


Fig. 1. A contrived implementation of the quantum not operation using an ancilla

width of the circuit produced in output by `iter dumbNot n` is equal to 2, even though, overall, the number of qubits initialized during the computation is equal to `n`. The point is that each ancilla is created only *after* the previous one has been discarded, thus enabling a form of qubit recycling.

```
iter :: (Qubit -> Circ Qubit)
-> Int -> Qubit -> Circ Qubit
iter f 0 q = return q
iter f n q = do
 q <- f q
 iter f (n-1) q
```

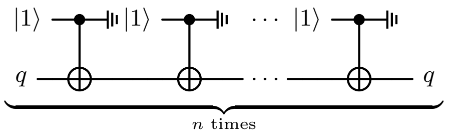


Fig. 2. A higher-order function which iterates a circuit-building function `f` on an input qubit `q` and the result of its application to the `dumbNot` function from Figure 1

Is it possible to statically analyze the width of the circuit produced in output by `iter dumbNot n` so as to conclude that it is constant and equal to 2? What techniques can we use? Certainly, the presence of higher order types complicates an already non-trivial problem. The approach we propose in this paper is based on two ingredients. The first is the so-called effect typing [40]. In this context the effect produced by the program is nothing more than the circuit and therefore it is natural to think of an effect system in which the width of such circuit, and only that, is exposed. Therefore, the arrow type $A \to B$ should be decorated with an expression indicating the width of the circuit produced by the corresponding function when applied to an argument of type A. Of course, the width of an individual circuit is a natural number, so it would make sense to annotate the arrow with such a number. For technical reasons, however, it will also be necessary to keep track of another natural number, corresponding to the number of wire resources that the function captures from the surrounding environment. This necessity stems from a need to keep track of wires even in the presence of data hiding, and will be explained in further detail in Section 4.

Under these premises, the `dumbNot` function would receive type $\mathsf{Qubit} \to_{2,0} \mathsf{Qubit}$, meaning that it takes as input a qubit and produces a circuit of width 2 which outputs a qubit. Note that the second annotation is 0, since we do not capture anything from the function's environment, let alone a wire. Consequently, because `iter` iterates in sequence and because the ancillary qubit in `dumbNot` can be reused, the type of `iter dumbNot n` would also be $\mathsf{Qubit} \rightarrow_{2,0} \mathsf{Qubit}$.

```
hadamardN :: [Qubit] -> Circ [Qubit]
hadamardN [] = return []
hadamardN (q:qs) = do
 q <- hadamard q
 qs <- hadamardN qs
 return (q:qs)
```

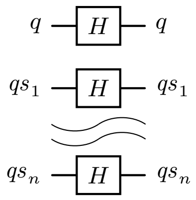


Fig. 3. The `hadamardN` function implements a circuit family where circuits have width linear in their input size.

Let us now consider a slightly different situation, in which the width of the produced circuit is not constant, but rather increases proportionally to the circuit's input size. Figure 3 shows a `Quipper` function that returns a circuit on n qubits in which the Hadamard gate is applied to each qubit, a common preprocessing step in many quantum algorithms. It is obvious that this function works on inputs of arbitrary size, and therefore we can interpret it as a circuit family, parametrized on the length of the input list of qubits. This quantity, although certainly a natural number, is unknown statically and corresponds precisely to the width of the produced circuit. It is thus natural to wonder whether the kind of effect typing we briefly hinted at in the previous paragraph is capable of dealing with such a function. Certainly, the expressions used to annotate arrows cannot be, like in the previous case, mere *constants*, as they clearly depend on the size of the input list. Is there a way to reflect this dependency in types? Certainly, one could go towards a fully-fledged notion of dependent types, like the ones proposed in [22], but a simpler approach, in the style of Dal Lago and Gaboardi's linear dependent types [12,14,15,23] turns out to be enough for this purpose. This is precisely the route that we follow in this paper. In this approach, terms can indeed appear in types, but that is only true for a very restricted class of terms, disjoint from the ordinary ones, called *index terms*. As an example, the type of the function `hadamardN` above could become $\mathsf{List}^i\ \mathsf{Qubit} \rightarrow_{i,0} \mathsf{List}^i\ \mathsf{Qubit}$, where i is an *index variable*. The meaning of the type would thus be that `hadamardN` takes as input any list of qubits of length i and produces a circuit of width at most i which outputs i qubits. Indices are better explained in Section 4, but in general we can say that they consist of arithmetical expressions over natural numbers and index variables, and can thus express non-trivial dependencies between input sizes and corresponding circuit widths.

3 The Proto-Quipper Language

This section aims at introducing the Proto-Quipper family of calculi to the non-specialist, without any form of resource analysis. At its core, Proto-Quipper is a linear lambda calculus with bespoke constructs to build and manipulate circuits. Circuits are built as a side-effect of a computation, behind the scenes, but they can also appear and be manipulated as data in the language.

Types	$TYPE\ \ A, B$	$::= \mathbb{1} \mid w \mid\ !A \mid A \otimes B \mid A \multimap B \mid \mathsf{List}\ A \mid \mathsf{Circ}(T, U)$
Parameter types	$PTYPE\ \ P, R$	$::= \mathbb{1} \mid\ !A \mid P \otimes R \mid \mathsf{List}\ P \mid \mathsf{Circ}(T, U)$
Bundle types	$BTYPE\ \ T, U$	$::= \mathbb{1} \mid w \mid T \otimes U$

Fig. 4. The Proto-Quipper types

The types of Proto-Quipper are given in Figure 4. Speaking at a high level, we can say that Proto-Quipper employs a linear-nonlinear typing discipline. In particular, $w \in \{\mathsf{Bit}, \mathsf{Qubit}\}$ is a *wire type* and is linear, while $\multimap$ is the linear arrow constructor. A subset of types, called *parameter types*, represent the values of the language that are *not* linear and that can therefore be copied. Any term of type A can be *lifted* into a duplicable parameter of type $!A$ if its type derivation does not require the use of linear resources.

Terms	$TERM\ \ M, N$	$::= V\,W \mid \mathsf{let}\ \langle x, y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\, V \mid \mathsf{box}_T\, V$
		$\mid \mathsf{apply}(V, W) \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$
Values	$VAL\ \ V, W$	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\bar{\ell}, \mathcal{C}, \bar{k}) \mid \langle V, W\rangle$
		$\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}\ V\ W$
Wire bundles	$BVAL\ \ \bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle$

Fig. 5. The Proto-Quipper syntax

The syntax of Proto-Quipper is given in Figure 5. At a very high level, we are dealing with an effectful lambda calculus with bespoke constructs for manipulating circuits. A return expression turns a value into a trivial computation, while a let expression is used to sequence computations. Note that let is associative and that return acts as its identity. Now, let us informally dissect the domain-specific aspects of this language, starting with the language of values. The constructs of greatest interest are *labels* and *boxed circuits*. A label ℓ represents a reference to a free wire of the underlying circuit being built and is attributed a wire type $w \in \{\mathsf{Bit}, \mathsf{Qubit}\}$. Due to the no-cloning property of quantum states [39], labels have to be treated linearly. Arbitrary structures of labels form a subset of values which we call *wire bundles* and which are given *bundle types*. On the other hand, a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ represents a circuit object $\mathcal{C}$ as a datum within the language, together with its input and output interfaces $\bar{\ell}$ and $\bar{k}$. Such a value is given parameter type $\mathsf{Circ}(T, U)$, where bundle types T and U are the input and output types of the circuit, respectively. Boxed circuits can be copied, manipulated by primitive functions and, more importantly, applied to the underlying circuit. This last operation, which lies at the core of Proto-Quipper's circuit-building capabilities, is possible thanks to the apply operator. This operator takes as first argument a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ and appends $\mathcal{C}$ to the underlying circuit $\mathcal{D}$. How does apply know *where* exactly in $\mathcal{D}$ to apply $\mathcal{C}$? Thanks to a second argument: a bundle of wires $\bar{t}$ coming from the free output wires of $\mathcal{D}$, which identify the exact location where $\mathcal{C}$ is supposed to be attached.

The language is expected to be endowed with constant boxed circuits corresponding to fundamental gates and operations (e.g. Hadamard, CNOT, initialization, etc.), but the programmer can also introduce their own boxed circuits via the box operator. Intuitively, box takes as input a circuit-building function and executes it in a sandboxed environment, on dummy arguments, in a way that leaves the underlying circuit unchanged. Said function produces a standalone circuit $\mathcal{C}$, which is then returned by the box operator as a boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$.

Figure 6 shows the Proto-Quipper term corresponding to the `Quipper` program in Figure 1, as an example of the use of the language. Note that $\mathsf{let}\ \langle x, y\rangle = M\ \mathsf{in}\ N$ is syntactic sugar for $\mathsf{let}\ z = M\ \mathsf{in}\ \mathsf{let}\ \langle x, y\rangle = z\ \mathsf{in}\ N$. The *dumbNot* function is given type $\mathsf{Qubit} \multimap \mathsf{Qubit}$ and builds the circuit shown in Figure 1 when applied to an argument.

$$\begin{aligned}
\mathit{dumbNot} \triangleq \lambda q_{\mathsf{Qubit}}.\ & \mathsf{let}\ a = \mathsf{apply}(\mathsf{INIT}_1, *)\ \mathsf{in}\\
& \mathsf{let}\ \langle q, a\rangle = \mathsf{apply}(\mathsf{CNOT}, \langle q, a\rangle)\ \mathsf{in}\\
& \mathsf{let}\ \_ = \mathsf{apply}(\mathsf{DISCARD}, a)\ \mathsf{in}\\
& \mathsf{return}\ q
\end{aligned}$$

Fig. 6. An example Proto-Quipper program. $\mathsf{INIT}_1, \mathsf{CNOT}$ and $\mathsf{DISCARD}$ are primitive boxed circuits implementing the corresponding elementary operations.

On the classical side of things, it is worth mentioning that Proto-Quipper as presented in this section does *not* support general recursion. A limited form of recursion on lists is instead provided via a primitive fold constructor, which takes as argument a (copiable) step function of type $!((B \otimes A) \multimap B)$, an initial value of type B, and constructs a function of type $\mathsf{List}\ A \multimap B$. Although this workaround is not enough to recover the full power of general recursion, it appears that it is enough to describe many quantum algorithms. Figure 7 shows an example of the use of fold to reverse a list. Note that $\lambda\langle x, y\rangle_{A\otimes B}.M$ is syntactic sugar for $\lambda z_{A\otimes B}.\mathsf{let}\ \langle x, y\rangle = z\ \mathsf{in}\ M$.

$$rev \triangleq \mathsf{fold}\ \mathsf{lift}(\lambda\langle revList, q\rangle_{\mathsf{List\ Qubit}\otimes\mathsf{Qubit}}.\mathsf{return}\ (\mathsf{cons}\ q\ revList))\ \mathsf{nil}$$

Fig. 7. An example of the use of fold: the function that reverses a list

To conclude this section, we just remark how all of the `Quipper` programs shown in Section 2 can be encoded in Proto-Quipper. However, Proto-Quipper's system of simple types in unable to tell us anything about the resource consumption of these programs. Of course, one could run `hadamardN` on a concrete input and examine the size of the circuit produced at run-time, but this amounts to *testing*, not *verifying* the program, and lacks the qualities of staticity and parametricity that we seek.

4 Incepting Linear Dependency and Effect Typing

We are now ready to expand on the informal definition of the Proto-Quipper language given in Section 3, to reach a formal definition of Proto-Quipper-R: a linearly and dependently typed language whose type system supports the derivation of upper bounds on the width of the circuits produced by programs.

4.1 Types and Syntax of Proto-Quipper-R

Types	*TYPE*	A, B	$::= \mathbb{1} \mid w \mid {!}A \mid A \otimes B \mid A \multimap_{I,J} B \mid \mathsf{List}^I A \mid \mathsf{Circ}^I(T,U)$
Param. types	*PTYPE*	P, R	$::= \mathbb{1} \mid {!}A \mid P \otimes R \mid \mathsf{List}^I P \mid \mathsf{Circ}^I(T,U)$
Bundle types	*BTYPE*	T, U	$::= \mathbb{1} \mid w \mid T \otimes U \mid \mathsf{List}^I T$
Terms	*TERM*	M, N	$::= V\,W \mid \mathsf{let}\ \langle x,y\rangle = V\ \mathsf{in}\ M \mid \mathsf{force}\,V \mid \mathsf{box}_T\,V$
			$\mid \mathsf{apply}(V,W) \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$
Values	*VAL*	V, W	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\,M \mid (\bar{\ell}, \mathcal{C}, \bar{k}) \mid \langle V, W\rangle$
			$\mid \mathsf{nil} \mid \mathsf{cons}\ V\ W \mid \mathsf{fold}_i\ V\ W$
Wire bundles	*BVAL*	$\bar{\ell}, \bar{k}$	$::= * \mid \ell \mid \langle \bar{\ell}, \bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{cons}\ \bar{\ell}\ \bar{k}$
Indices	*INDEX*	I, J	$::= i \mid n \mid I + J \mid I - J \mid I \times J \mid \max(I,J) \mid \max_{i<I} J$

Fig. 8. Proto-Quipper-R syntax and types

The types and syntax of Proto-Quipper-R are given in Figure 8. As we mentioned, one of the key ingredients of our type system are the index terms with which we annotate standard Proto-Quipper types. These indices provide quantitative information about the elements of the resulting types, in a manner reminiscent of refinement types [18,47]. In our case, we are primarily concerned with circuit width, which means that the natural starting point of our extension of Proto-Quipper is precisely the circuit type: $\mathsf{Circ}^I(T, U)$ has elements the boxed circuits of input type T, output type U, *and width bounded by I*. Term I is precisely what we call an index, that is, an arithmetical expression denoting a natural number. Looking at the grammar for indices, their interpretation is fairly straightforward, with a few notes: n is a natural number, i is an index variable, $I - J$ denotes *natural* subtraction, such that $I - J = 0$ whenever $I \leq J$, and lastly $\mathsf{max}_{i<I} J$ is the maximum for i going from 0 (included) to I (excluded) of J, where i can occur in J. Note that $I = 0$ implies $\mathsf{max}_{i<I} J = 0$. While the index in a circuit type denotes an upper bound, the index in a type of the form $\mathsf{List}^I A$ denotes the *exact* length of the lists of that type. While this refinement might seem quite restrictive in a generic scenario, it allows us to include lists of labels among wire bundles, something that was not possible with simple lists. This is due to the fact that sized lists are effectively isomorphic to finite tensors, and therefore a sized list of labels represent a wire bundle of known size, whereas the same is not true for a simple list of labels. Lastly, as we anticipated in Section 2, an arrow type $A \multimap_{I,J} B$ is annotated with *two* indices: I is an upper bound to the width of the circuit built by the function once it is applied to an argument of type A, while J describes the exact number of wires captured in the function's closure. The utility of this last annotation will be clearer in Section 4.3.

The languages for terms and values are almost the same as in Proto-Quipper, with the minor difference that the fold operator now binds the index variable name i within its first argument. This variable appears locally in the type of the step function, in such a way as to allow each iteration of the fold to contribute to the overall circuit width in a *different* way.

4.2 A Formal Language for Circuits

The type system of Proto-Quipper-R is designed to allow for reasoning about the width of circuits. Therefore, before we formally introduce the type system in Section 4.3, we ought to introduce circuits themselves in a formal way. So far, we have only spoken of circuits at a very high and intuitive level, and we have represented them only graphically. Looking at the circuits in Section 2, what do they have in common? At the fundamental level, they are made up of elementary operations applied to specific wires. Of course, the order of these operations matters, as does the order of wires that they are applied to. In the existing literature on Proto-Quipper, circuits are usually interpreted as morphisms in a symmetric monoidal category [46], but this approach makes it particularly hard to reason about their intensional properties, such as width. For this reason, we opt for a *concrete* model of wires and circuits, rather than an abstract one.

Luckily, we already have a datatype modeling ordered structures of wires, that is, the wire bundles that we introduced in the previous sections. We use them as the basis upon which we build circuits.

That being said, Figure 9 introduces the Circuit Representation Language (CRL) which we use as the target for circuit building in Proto-Quipper-R. Wire bundles are exactly as in Figure 8 and represent arbitrary structures of wires,

Wire bundles	$BVAL$	$\bar{\ell},\bar{k} ::= * \mid \ell \mid \langle \bar{\ell},\bar{k}\rangle \mid \mathsf{nil} \mid \mathsf{cons}\,\bar{\ell}\,\bar{k}$
Bundle types	$BTYPE$	$T,U ::= \mathbb{1} \mid w \mid T \otimes U \mid \mathsf{List}^{I}\,T$
Circuits	$CIRC$	$\mathcal{C},\mathcal{D} ::= id_Q \mid \mathcal{C}; g(\bar{\ell}) \to \bar{k}$

Fig. 9. CRL syntax and types

while circuits themselves are defined very simply as sequences of elementary operations applied to said structures. We call Q a *label context* and define it as a mapping from label names to wire types. We use label contexts as a means to keep track of the set of labels available in a computation, alongside their respective types. Circuit id_Q represents the identity circuit taking as input the labels in Q and returning them unchanged, while $\mathcal{C}; g(\bar{\ell}) \to \bar{k}$ represents the application of the elementary operation g to the wires identified by $\bar{\ell}$ among the outputs of $\mathcal{C}$. Operation g outputs the wire bundle $\bar{k}$, whose labels become part of the outputs of the overall circuit. Note that an "elementary operation" is usually the application of a gate, but it could also be a measurement, or the initialization or discarding of a wire. Although semantically very different, from the perspective of circuit building these operations are just elementary building blocks in the construction of a more complex structure, and it makes no sense to distinguish between them syntactically. Circuits are amenable to a form of concatenation. We write the *concatenation* of $\mathcal{C}$ and $\mathcal{D}$ as $\mathcal{C} :: \mathcal{D}$ and define it in the natural way, that is, as $\mathcal{C}$ followed by all the operations occurring in $\mathcal{D}$.

Circuit Typing Naturally, not all circuits built from the CRL syntax make sense. For example $id_{(\ell:\mathsf{Qubit})}; H(k) \to k$ and $id_{(\ell:\mathsf{Qubit})}; CNOT(\langle \ell, \ell\rangle) \to \langle k, t\rangle$ are both syntactically correct, but the first applies a gate to a non-existing wire, while the second violates the no-cloning theorem by duplicating ℓ. To rule out such ill-formed circuits, we employ a rudimentary type system for circuits which allows us to derive judgments of the form $\mathcal{C} : Q \to L$, which informally read "circuit $\mathcal{C}$ is well-typed with input label context Q and output label context L".

The typing rules for CRL are given in Figure 10. We call $Q \vdash_w \bar{\ell} : T$ a *wire judgment*, and we use it to give a structured type to an otherwise unordered label context, by means of a wire bundle. Most rules are straightforward, except those for lists, which rely on a judgment of the form $\vDash I = J$. This is to be intended as a semantic judgment asserting that I and J are closed and equal when interpreted as natural numbers. Within the rule, this reflects the idea that there are many ways to syntactically represent the length of a list. For example, nil can be given type $\mathsf{List}^0\,T$, but also $\mathsf{List}^{1-1}\,T$ or $\mathsf{List}^{0\times 5}\,T$. This kind of flexibility might seem unwarranted for such a simple language, but it is useful to effectively interface CRL and the more complex Proto-Quipper-R. Speaking of the actual circuit judgments, the *seq* rule tells us that the the application of an elementary operation g is well typed whenever g only acts on labels occurring in the outputs of $\mathcal{C}$ (those in $\bar{\ell}$, that is in H), produces in output labels that do not clash with the remaining outputs of $\mathcal{C}$ (since L, K denotes the disjoint

$$
\mathit{unit}\,\frac{}{\emptyset \vdash_w * : \mathbb{1}}
\qquad
\mathit{lab}\,\frac{}{\ell : w \vdash_w \ell : w}
\qquad
\mathit{nil}\,\frac{\vDash I = 0}{\emptyset \vdash_w \mathsf{nil} : \mathsf{List}^I\, T}
$$

$$
\mathit{pair}\,\frac{Q_1 \vdash_w \bar{\ell} : T \qquad Q_2 \vdash_w \bar{k} : U}{Q_1, Q_2 \vdash_w \langle \bar{\ell}, \bar{k} \rangle : T \otimes U}
$$

$$
\mathit{cons}\,\frac{Q_1 \vdash_w \bar{\ell} : T \qquad Q_2 \vdash_w \bar{k} : \mathsf{List}^J\, T \qquad \vDash I = J + 1}{Q_1, Q_2 \vdash_w \mathsf{cons}\; \bar{\ell}\; \bar{k} : \mathsf{List}^I\, T}
$$

$$
\mathit{id}\,\frac{}{\mathit{id}_Q : Q \to Q}
\qquad
\mathit{seq}\,\frac{\mathcal{C} : Q \to L, H \quad H \vdash_w \bar{\ell} : T \quad K \vdash_w \bar{k} : U \quad g \in \mathscr{G}(T, U)}{\mathcal{C}; g(\bar{\ell}) \to \bar{k} : Q \to L, K}
$$

Fig. 10. The CRL type system

union of the two label contexts) and is of the right type. This last requirement is expressed as $g \in \mathscr{G}(T, U)$, where $\mathscr{G}(T, U)$ is the subset of elementary operations that can be applied to an input of type T to obtain an output of type U. For example, the Hadamard gate, which acts on a single qubit, is in $\mathscr{G}(\mathsf{Qubit}, \mathsf{Qubit})$.

Circuit Width Among the many properties of circuits, we are interested in width, so we conclude this section by giving a formal status to this quantity.

Definition 1 (Circuit Width). *We define the width of a CRL circuit $\mathcal{C}$, written* $\mathrm{width}(\mathcal{C})$*, as follows*

$$
\begin{aligned}
\mathrm{width}(\mathit{id}_Q) &= |Q|, && (1)\\
\mathrm{width}(\mathcal{C}; g(\bar{\ell}) \to \bar{k}) &= \mathrm{width}(\mathcal{C}) + \max(0, \mathrm{new}(g) - \mathrm{discarded}(\mathcal{C})), && (2)
\end{aligned}
$$

where $|Q|$ is the number of labels in Q, $\mathrm{new}(g)$ represents the net number of new wires initialized by g, and $\mathrm{discarded}(\mathcal{C})$ is the number of wires that have been effectively discarded by the end of $\mathcal{C}$, obtained as the difference between $\mathcal{C}$'s width and the number of its outputs. Note that one expects $\mathrm{new}(g)$ to be equal to the difference between the number of labels in $\bar{k}$ and those in $\bar{\ell}$. The overarching idea behind this definition is that whenever we require new wires in our computation, we first try to reuse as many previously discarded wires as possible. As long as we can do this ($\mathrm{new}(g) \leq \mathrm{discarded}(\mathcal{C})$), the initializations do not add to the total width of the circuit. Otherwise ($\mathrm{new}(g) > \mathrm{discarded}(\mathcal{C})$) we must actually create new wires, increasing the overall width of the circuit.

Now that we have a formal definition of circuit types and width, we can state a fundamental property of the concatenation of well-typed circuits, which is illustrated in Figure 11 and proven in Theorem 1. We use this result pervasively in proving the correctness of Proto-Quipper-R in section 5.

Theorem 1 (CRL). *Given $\mathcal{C} : Q \to L, H$ and $\mathcal{D} : H \to K$ such that the labels shared by $\mathcal{C}$ and $\mathcal{D}$ are all and only those in H, we have*

1. $\mathcal{C} :: \mathcal{D} : Q \to L, K$, *2.* $\mathrm{width}(\mathcal{C} :: \mathcal{D}) \leq \max(\mathrm{width}(\mathcal{C}), \mathrm{width}(\mathcal{D}) + |L|)$.

Proof. By induction of the derivation of $\mathcal{D} : H \to K$.

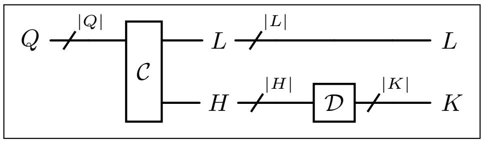


Fig. 11. The kind of scenario described by Theorem 1

4.3 Typing Programs

Going back to Proto-Quipper-R, we have already seen how the standard Proto-Quipper types are refined with quantitative information. However, decorating types is not enough for the purposes of width estimation. Recall that, in general, a Proto-Quipper program produces a circuit as a *side effect* of its evaluation. If we want to reason about the width of said circuit, it is not enough to rely on a regular linear type system, although dependent. Rather, we have to introduce the second ingredient of our analysis and turn to a *type-and-effect system* [40], revolving around a type judgment of the form

$$\Theta; \Gamma; Q \vdash_c M : A; I, \tag{3}$$

which intuitively reads "for all natural values of the index variables in Θ, under typing context Γ and label context Q, term M has type A and produces a circuit of width at most I". Therefore, Θ is a collection of index variables which are universally quantified in the rest of the judgment, while Γ is a typing context for parameter and linear variables alike. When a typing context contains exclusively parameter variables, we write it as Φ. In this judgment, I plays the role of an *effect annotation*, describing a relevant aspect of the side effect produced by the evaluation of M (i.e. the width of the produced circuit). The attentive reader might wonder why this annotation consists only of one index, whereas when we discussed arrow types in previous sections we needed two. The reason is that the second index, which we use to keep track of the number of wires captured by a function, is redundant in a typing judgment where the same quantity can be inferred directly from the environments Γ and Q. A similar typing judgment of the form $\Theta; \Gamma; Q \vdash_v V : A$ is introduced for values, which are effect-less.

The rules for deriving typing judgments are those in Figure 12, where Γ_1, Γ_2 denotes the union of two contexts with disjoint domains. A well-formedness judgment of the form $\Theta \vdash I$ means that all the free index variables occurring in I are in Θ. Well-formedness is lifted to types and typing contexts in the

$$\mathit{unit}\,\frac{\Theta \vdash \Phi}{\Theta; \Phi; \emptyset \vdash_v * : \mathbb{1}} \qquad \mathit{lab}\,\frac{\Theta \vdash \Phi}{\Theta; \Phi; \ell : w \vdash_v \ell : w}$$

$$\mathit{var}\,\frac{\Theta \vdash \Phi, x : A}{\Theta; \Phi, x : A; \emptyset \vdash_v x : A} \qquad \mathit{abs}\,\frac{\Theta; \Gamma, x : A; Q \vdash_c M : B; I}{\Theta; \Gamma; Q \vdash_v \lambda x_A.M : A \multimap_{I,\#(\Gamma;Q)} B}$$

$$\mathit{app}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : A \multimap_{I,J} B \qquad \Theta; \Phi, \Gamma_2; Q_2 \vdash_v W : A}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c V\,W : B; I}$$

$$\mathit{lift}\,\frac{\Theta; \Phi; \emptyset \vdash_c M : A; 0}{\Theta; \Phi; \emptyset \vdash_v \mathsf{lift}\, M : !\,A} \qquad \mathit{force}\,\frac{\Theta; \Phi; \emptyset \vdash_v V : !\,A}{\Theta; \Phi; \emptyset \vdash_c \mathsf{force}\, V : A; 0}$$

$$\mathit{circ}\,\frac{\mathcal{C} : Q \to L \qquad Q \vdash_w \bar{\ell} : T \qquad L \vdash_w \bar{k} : U \qquad \Theta \vDash \mathrm{width}(\mathcal{C}) \leq I \qquad \Theta \vdash \Phi}{\Theta; \Phi; \emptyset \vdash_v (\bar{\ell}, \mathcal{C}, \bar{k}) : \mathsf{Circ}^I(T, U)}$$

$$\mathit{apply}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : \mathsf{Circ}^I(T, U) \qquad \Theta; \Phi, \Gamma_2; Q_2 \vdash_v W : T}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c \mathsf{apply}(V, W) : U; I}$$

$$\mathit{box}\,\frac{\Theta; \Phi; \emptyset \vdash_v V : !(T \multimap_{I,J} U)}{\Theta; \Phi; \emptyset \vdash_c \mathsf{box}_T\, V : \mathsf{Circ}^I(T, U); 0} \qquad \mathit{nil}\,\frac{\Theta \vdash \Phi \qquad \Theta \vdash A}{\Theta; \Phi; \emptyset \vdash_v \mathsf{nil} : \mathsf{List}^0\, A}$$

$$\mathit{cons}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : A \qquad \Theta; \Phi, \Gamma_2; Q_2 \vdash_v W : \mathsf{List}^I\, A}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1 Q_2 \vdash_v \mathsf{cons}\ V\ W : \mathsf{List}^{I+1}\, A}$$

$$\mathit{fold}\,\frac{\begin{array}{c}\Theta; \Phi, \Gamma; Q \vdash_v W : B\{0/i\} \qquad \Theta, i; \Phi; \emptyset \vdash_v V : !((B \otimes A) \multimap_{J,J'} B\{i+1/i\}) \\ \Theta \vdash I \qquad \Theta \vdash A \qquad E = \mathsf{max}(\#(\Gamma; Q), \mathsf{max}_{i<I}\, J + (I - 1 - i) \times \#(A))\end{array}}{\Theta; \Phi, \Gamma; Q \vdash_v \mathsf{fold}_i\ V\ W : \mathsf{List}^I\, A \multimap_{E,\#(\Gamma;Q)} B\{I/i\}}$$

$$\mathit{dest}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : A \otimes B \qquad \Theta; \Phi, \Gamma_2, x : A, y : B; Q_2 \vdash_c M : C; I}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c \mathsf{let}\ \langle x, y \rangle = V\ \mathsf{in}\ M : C; I}$$

$$\mathit{pair}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : A \qquad \Theta; \Phi, \Gamma_2; Q_2 \vdash_v W : B}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_v \langle V, W \rangle : A \otimes B}$$

$$\mathit{return}\,\frac{\Theta; \Gamma; Q \vdash_v V : A}{\Theta; \Gamma; Q \vdash_c \mathsf{return}\ V : A; \#(\Gamma; Q)}$$

$$\mathit{let}\,\frac{\Theta; \Phi, \Gamma_1; Q_1 \vdash_c M : A; I \qquad \Theta; \Phi, \Gamma_2, x : A; Q_2 \vdash_c N : B; J}{\Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c \mathsf{let}\ x = M\ \mathsf{in}\ N : B; \mathsf{max}(I + \#(\Gamma_2; Q_2), J)}$$

$$\mathit{vsub}\,\frac{\Theta; \Gamma; Q \vdash_v V : A \qquad \Theta \vdash_s A <: B}{\Theta; \Gamma; Q \vdash_v V : B}$$

$$\mathit{csub}\,\frac{\Theta; \Gamma; Q \vdash_c M : A; I \qquad \Theta \vdash_s A <: B \qquad \Theta \vDash I \leq J}{\Theta; \Gamma; Q \vdash_c M : B; J}$$

Fig. 12. Proto-Quipper-R type system

natural way. Among interesting typing rules, we can see how the *circ* rule bridges between CRL and Proto-Quipper-R. A boxed circuit $(\bar{\ell}, \mathcal{C}, \bar{k})$ is well typed with type $\mathsf{Circ}^I(T, U)$ when $\mathcal{C}$ is no wider than the quantity denoted by I, $\mathcal{C} : Q \to L$ and $\bar{\ell}, \bar{k}$ contain all and only the labels in Q and L, respectively, acting as a language-level interface to $\mathcal{C}$.

The two main constructs that interact with circuits are apply and box. The *apply* rule is the foremost place where effects enter the type derivation: V represents some boxed circuit of width at most I, so its application to an appropriate wire bundle W produces exactly a circuit of width at most I. The *box* rule, on the other hand, works approximately in the opposite direction. If V is a circuit building function that, once applied to an input of type T, would build a circuit of output type U and width at most I, then boxing it means turning it into a boxed circuit with the same characteristics. Note that the *box* rule requires that the typing context be devoid of linear variables. This reflects the idea that V is meant to be executed in complete isolation, to build a standalone, replicable circuit, and therefore it should not capture any linear resource (e.g. a label) from the surrounding environment.

Wire Count Notice that many rules rely on an operator written $\#(\cdot)$, which we call the *wire count* operator. Intuitively, this operator returns the number of wire resources (in our case, bits or qubits) represented by a type or context. To understand how this is important, consider the *return* rule. The return operator turns a value V into a trivial computation that evaluates immediately to V, and therefore it would be tempting to give it an effect annotation of 0. However, V is not necessarily a closed value. In fact, it might very well contain many bits and qubits, coming both from the typing context Γ and the label context Q. Although nothing happens to these bits and qubits, they still corresponds to wires in the underlying circuit, and these wires have a width which must be accounted for in the judgment for the otherwise trivial computation. The *return* rule therefore produces an effect annotation of the form $\#(\Gamma; Q)$, which is shorthand for $\#(\Gamma) + \#(Q)$ and corresponds exactly to this quantity. A formal definition of the wire count operator on types is given in the following definition, which is lifted to contexts in the natural way.

Definition 2 (Wire Count). *We define the* wire count *of a type A, written $\#(A)$, as a function $\#(\cdot) : TYPE \to INDEX$ such that*

$$\#(\mathbb{1}) = \#(!\,A) = \#(\mathsf{Circ}^I(T, U)) = 0, \qquad \#(w) = 1,$$

$$\#(A \otimes B) = \#(A) + \#(B), \qquad \#(A \multimap_{I,J} B) = J, \qquad \#(\mathsf{List}^I\, A) = I \times \#(A).$$

This definition is fairly straightforward, except for the arrow case. By itself, an arrow type does not give us any information about the amount of qubits or bits captured in the corresponding closure. This is precisely where the second index J, which keeps track exactly of this quantity, comes into play. This annotation is introduced by the *abs* rule and allows our analysis to circumvent data hiding.

The *let* rule is another rule in which wire counts are essential. The two terms M and N in $\mathsf{let}\ x = M\ \mathsf{in}\ N$ build the circuits $\mathcal{C}_M$ and $\mathcal{C}_N$, whose widths are bounded by I and J, respectively. Once again, it might be tempting to conclude that the overall circuit built by the let construct has width bounded by $\mathsf{max}(I, J)$, but this fails to take into account the fact that while M is building $\mathcal{C}_M$ starting from the wires contained in Γ_1 and Q_1, we must keep aside the wires contained in Γ_2 and Q_2, which will be used by N to build $\mathcal{C}_N$. These wires must flow alongside $\mathcal{C}_M$ and their width, i.e. $\#(\Gamma_2; Q_2)$, adds up to the total width of the left-hand side of the let construct, leading to an overall width upper bound of $\mathsf{max}(I + \#(\Gamma_2; Q_2), J)$. This situation is better illustrated in Figure 13.

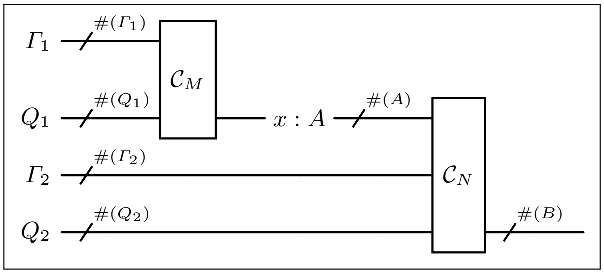


Fig. 13. The shape of a circuit built by a let construct

The last rule that makes substantial use of wire counts is *fold*, arguably the most complex rule of the system. The main ingredient of the fold rule is the bound index variable i, which occurs in the accumulator type B and is used to keep track of the number of steps performed by the fold. Let $(\cdot)\{I/i\}$ denote the capture-avoiding substitution of the index term I for the index variable i inside an index, type, context, value or term, not unlike $(\cdot)[V/x]$ denotes the capture-avoiding substitution of the value V for the variable x. Intuitively, if the accumulator has initially type $B\{0/i\}$ and each application of the step function increases i by one, then when we fold over a list of length I we get an output of type $B\{I/i\}$. Index E is the upper bound to the width of the overall circuit built by the fold: if the input list is empty, then the width of the circuit is just the number of wires contained in the initial accumulator, that is, $\#(\Gamma; Q)$. If the input list is non-empty, on the other hand, things get slightly more complicated. At each step i, the step function builds a circuit $\mathcal{C}_i$ of width bounded by J, where J might depend on i. This circuit takes as input all the wires in the accumulator, as well as the wires contained in the first element of the input list, which are $\#(A)$. The wires contained in remaining $I-1-i$ elements have to flow alongside $\mathcal{C}_i$, giving a width upper bound of $J + (I - 1 - i) \times \#(A)$ at each step i. The overall width upper bound is then the maximum for i going from 0 to $I - 1$ of this quantity, i.e. precisely $\mathsf{max}_{i<I}\, J + (I-1-i) \times \#(A)$. Once again, a graphical representation of this scenario is given in Figure 14.

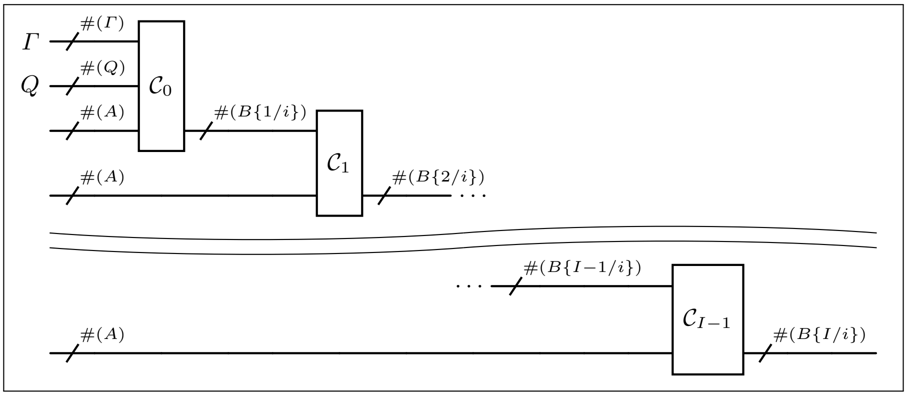


Fig. 14. The shape of a circuit built by a fold applied to an input list of type $\mathsf{List}^I\ A$

Subtyping Notice that Proto-Quipper-R's type system includes two subsumption rules, which are effectively the same rule for terms and values, respectively: *csub* and *vsub*. We mentioned that our type system resembles a refinement type system, and all such systems induce a subtyping relation between types, where A is a subtype of B whenever the former is "at least as refined" as the latter. In our case, a subtyping judgment such as $\Theta \vdash_s A <: B$ means that for all natural values of the index variables in Θ, A is a subtype of B.

$$\textit{unit}\,\frac{}{\Theta \vdash_s \mathbb{1} <: \mathbb{1}} \qquad \textit{wire}\,\frac{}{\Theta \vdash_s w <: w} \qquad \textit{bang}\,\frac{\Theta \vdash_s A <: B}{\Theta \vdash_s\, !A <: !B}$$

$$\textit{tensor}\,\frac{\Theta \vdash_s A_1 <: A_2 \qquad \Theta \vdash_s B_1 <: B_2}{\Theta \vdash_s A_1 \otimes B_1 <: A_2 \otimes B_2}$$

$$\textit{arrow}\,\frac{\Theta \vdash_s A_2 <: A_1 \qquad \Theta \vdash_s B_1 <: B_2 \qquad \Theta \vDash I_1 \leq I_2 \qquad \Theta \vDash J_1 = J_2}{\Theta \vdash_s A_1 \multimap_{I_1,J_1} B_1 <: A_2 \multimap_{I_2,J_2} B_2}$$

$$\textit{list}\,\frac{\Theta \vdash_s A <: B \qquad \Theta \vDash I = J}{\Theta \vdash_s \mathsf{List}^I\ A <: \mathsf{List}^J\ B}$$

$$\textit{circ}\,\frac{\Theta \vdash_s T_1 <:> T_2 \qquad \Theta \vdash_s U_1 <:> U_2 \qquad \Theta \vDash I \leq J}{\Theta \vdash_s \mathsf{Circ}^I(T_1, U_1) <: \mathsf{Circ}^J(T_2, U_2)}$$

Fig. 15. Proto-Quipper-R subtyping rules

We derive this kind of judgments by the rules in Figure 15. Note that $\Theta \vdash_s A <:> B$ is shorthand for "$\Theta \vdash_s A <: B$ and $\Theta \vdash_s B <: A$". Subtyping relies in turn on a judgment of the form $\Theta \vDash I \leq J$, which is a generalization of the semantic judgment that we used in the CRL type system in Section 4.2. Such a judgment asserts that for all natural values of the index variables in Θ, I is lesser or equal than J. Consequently, $\Theta \vDash I = J$ is shorthand for "$\Theta \vDash I \leq J$ and $\Theta \vDash J \leq I$". We purposefully leave the decision procedure of this kind of judgments unspecified, with the prospect that, from a more practical perspective, they could be delegated to an SMT solver [7].

4.4 Operational Semantics

Operationally speaking, it does not make sense, in the Proto-Quipper languages, to speak of the semantics of a term *in isolation*: a term is always evaluated in the context of an underlying circuit that supplies all of the term's free labels. We therefore define the operational semantics of Proto-Quipper-R as a big-step evaluation relation $\Downarrow$ on *configurations*, i.e. circuits paired with either terms or values. Intuitively, $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$ means that M evaluates to V and updates $\mathcal{C}$ to $\mathcal{D}$ as a side effect.

$$\mathit{app}\,\frac{(\mathcal{C}, M[V/x]) \Downarrow (\mathcal{D}, W)}{(\mathcal{C}, (\lambda x_A.M)\, V) \Downarrow (\mathcal{D}, W)} \qquad \mathit{dest}\,\frac{(\mathcal{C}, M[V/x][W/y]) \Downarrow (\mathcal{D}, X)}{(\mathcal{C}, \mathsf{let}\ \langle x, y\rangle = \langle V, W\rangle\ \mathsf{in}\ M) \Downarrow (\mathcal{D}, X)}$$

$$\mathit{force}\,\frac{(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)}{(\mathcal{C}, \mathsf{force}(\mathsf{lift}\, M)) \Downarrow (\mathcal{D}, V)} \qquad \mathit{apply}\,\frac{(\mathcal{E}, \bar{q}) = \mathrm{append}(\mathcal{C}, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))}{(\mathcal{C}, \mathsf{apply}((\bar{\ell}, \mathcal{D}, \bar{k}), \bar{t})) \Downarrow (\mathcal{E}, \bar{q})}$$

$$\mathit{box}\,\frac{(Q, \bar{\ell}) = \mathrm{freshlabels}(T) \qquad (id_Q, M) \Downarrow (id_Q, V) \qquad (id_Q, V\,\bar{\ell}) \Downarrow (\mathcal{D}, \bar{k})}{(\mathcal{C}, \mathsf{box}_T(\mathsf{lift}\, M)) \Downarrow (\mathcal{C}, (\bar{\ell}, \mathcal{D}, \bar{k}))}$$

$$\mathit{return}\,\frac{}{(\mathcal{C}, \mathsf{return}\ V) \Downarrow (\mathcal{C}, V)} \qquad \mathit{let}\,\frac{(\mathcal{C}, M) \Downarrow (\mathcal{E}, V) \qquad (\mathcal{E}, N[V/x]) \Downarrow (\mathcal{D}, W)}{(\mathcal{C}, \mathsf{let}\ x = M\ \mathsf{in}\ N) \Downarrow (\mathcal{D}, W)}$$

$$\mathit{fold\text{-}end}\,\frac{}{(\mathcal{C}, (\mathsf{fold}_i\ V\ W)\ \mathsf{nil}) \Downarrow (\mathcal{C}, W)}$$

$$\mathit{fold\text{-}step}\,\frac{\begin{matrix}(\mathcal{C}, M\{0/i\}) \Downarrow (\mathcal{C}, Y) \qquad (\mathcal{C}, Y\,\langle V, W\rangle) \Downarrow (\mathcal{E}, Z) \\ (\mathcal{E}, (\mathsf{fold}_i\ (\mathsf{lift}\, M\{i+1/i\})\ Z)\ W') \Downarrow (\mathcal{D}, X)\end{matrix}}{(\mathcal{C}, (\mathsf{fold}_i\ (\mathsf{lift}\, M)\ V)\ (\mathsf{cons}\ W\ W')) \Downarrow (\mathcal{D}, X)}$$

Fig. 16. Proto-Quipper-R big-step operational semantics

The rules for evaluating configurations are given in Figure 16, where $\mathcal{C}, \mathcal{D}$ and $\mathcal{E}$ are circuits, M and N are terms, while V, W, X, Y and Z are values. Most evaluation rules are straightforward, with the exception perhaps of *apply, box* and *fold-step.* Being the fundamental block of circuit-building, the semantics of apply lies almost entirely in the way it updates the underlying circuit. The concatenation of the underlying circuit $\mathcal{C}$ and the applicand $\mathcal{D}$ is delegated entirely to the append function, which is given in Definition 4. Before we examine the append function, however, consider than when we deal with circuit objects we are not really interested in the concrete labels that occur in them, but rather in the *structure* that they convey. For this reason, we introduce the following notion of *circuit equivalence.*

Definition 3 (Circuit Equivalence). *We say that two boxed circuits* $(\bar{\ell}, \mathcal{C}, \bar{k})$ *and* $(\bar{t}, \mathcal{D}, \bar{q})$ *are* equivalent, *and we write* $(\bar{\ell}, \mathcal{C}, \bar{k}) \cong (\bar{t}, \mathcal{D}, \bar{q})$, *when there exists a renaming* ρ *of labels such that* $\rho(\bar{\ell}) = \bar{t}$, $\rho(\bar{k}) = \bar{q}$ *and* $\rho(\mathcal{C}) = \mathcal{D}$.

We can now move on to the definition of append, where the notion of circuit equivalence is used to instantiate the generic input interface of a boxed circuit with the actual labels that it is going to be appended to, and to ensure that there are no name clashes between the appended circuit and the underlying circuit.

Definition 4 (append**).** *We define* the append of $(\bar{\ell}, \mathcal{D}, \bar{k})$ to $\mathcal{C}$ on $\bar{t}$, *written* append$(\mathcal{C}, \bar{t}, (\bar{\ell}, \mathcal{D}, \bar{k}))$*, as the function that performs the following steps:*

1. *Finds* $(\bar{t}, \mathcal{D}', \bar{q}) \cong (\bar{\ell}, \mathcal{D}, \bar{k})$ *such that the labels shared by* $\mathcal{C}$ *and* $\mathcal{D}'$ *are all and only those in* $\bar{t}$*,*
2. *Computes* $\mathcal{E} = \mathcal{C} :: \mathcal{D}'$*,*
3. *Returns* $(\mathcal{E}, \bar{q})$*.*

On the other hand, the semantics of a term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$ relies on the freshlabels function. What freshlabels does is take as input a bundle type T and instantiate fresh $Q, \bar{\ell}$ such that $Q \vdash_w \bar{\ell} : T$. The wire bundle $\bar{\ell}$ is then used as a dummy argument to V, the circuit-building function resulting from the evaluation of M. This function application is evaluated in the context of the identity circuit id_Q and eventually produces a circuit $\mathcal{D}$, together with its output labels $\bar{k}$. Finally, $\bar{\ell}$ and $\bar{k}$ become respectively the input and output interfaces of the boxed circuit $(\bar{\ell}, \mathcal{D}, \bar{k})$, which is the result of the evaluation of $\mathsf{box}_T(\mathsf{lift}\, M)$.

Note, at this point, that T controls how many labels are initialized by the freshlabels function. Because T can contain indices (e.g. it could be that $T \equiv \mathsf{List}^3\, \mathsf{Qubit}$), it follows that in Proto-Quipper-R indices are not only relevant to typing, but they also have operational value. For this reason, the semantics of Proto-Quipper-R is well-defined only on terms closed both in the sense of regular variables *and* index variables, since a circuit-building function of input type, say, $\mathsf{List}^i\, \mathsf{Qubit}$ does not correspond to any individual circuit, and therefore it makes no sense to box it. This aspect of the semantics is also apparent in the *fold-step* rule, where the index variable i occurring free in M is instantiated to 0 before evaluating M to obtain the step function Y. Then, before evaluating the next fold, i is replaced with $i + 1$ in M, increasing the index for the next iteration.

5 Type Safety and Correctness

Because the operational semantics of Proto-Quipper-R is based on configurations, we ought to adopt a notion of well-typedness which is also based on configurations. The following definition of *well-typed configuration* is thus central to our type-safety analysis.

Definition 5 (Well-typed Configuration). *We say that configuration* $(\mathcal{C}, M)$ *is* well-typed with input Q, type A, width I and output L, *and we write* $Q \vdash (\mathcal{C}, M) : A; I; L$, *whenever* $\mathcal{C} : Q \to L, H$ *for some* H *such that* $\emptyset; \emptyset; H \vdash_c M : A; I$. *We write* $Q \vdash (\mathcal{C}, V) : A; L$ *whenever* $\mathcal{C} : Q \to L, H$ *for some* H *such that* $\emptyset; \emptyset; H \vdash_v V : A$.

The three results that we want to show in this section are that any well-typed term configuration $Q \vdash (\mathcal{C}, M) : A; I; L$ evaluates to some configuration $(\mathcal{D}, V)$, that $Q \vdash (\mathcal{D}, V) : A; L$ and that $\mathcal{D}$ is obtained from $\mathcal{C}$ by extending it with a sub-circuit of width at most I. These claims correspond to the *subject reduction* and *total correctness* properties that we will prove at the end of this section. However, both these results rely on a central lemma and on the mutual notions of *realization* and *reducibility*, which we first give formally.

Definition 6 (Realization). *We define* $V \Vdash_Q A$, *which reads* V realizes A under Q, *as the smallest relation such that*

- $* \Vdash_\emptyset \mathbb{1}$,
- $\ell \Vdash_{\ell:w} w$,
- $V \Vdash_Q A \multimap_{I,J} B$ *iff* $\vDash J = |Q|$ *and* $\forall W : W \Vdash_L A \implies V\,W \Vdash^I_{Q,L} B$,
- $\mathsf{lift}\, M \Vdash_\emptyset\, !A$ *iff* $M \Vdash^0_\emptyset A$,
- $\langle V, W \rangle \Vdash_{Q,L} A \otimes B$ *iff* $V \Vdash_Q A$ *and* $W \Vdash_L B$,
- $\mathsf{nil} \Vdash_\emptyset \mathsf{List}^I A$ *iff* $\vDash I = 0$,
- $\mathsf{cons}\; V\; W \Vdash_{Q,L} \mathsf{List}^I A$ *iff* $\vDash I = J + 1$ *and* $V \Vdash_Q A$ *and* $W \Vdash_L \mathsf{List}^J A$,
- $(\bar{\ell}, \mathcal{C}, \bar{k}) \Vdash_\emptyset \mathsf{Circ}^I(T, U)$ *iff* $\mathcal{C} : Q \to L$ *and* $Q \vdash_w \bar{\ell} : T$ *and* $L \vdash_w \bar{k} : U$ *and* $\vDash \mathrm{width}(\mathcal{C}) \leq I$.

Definition 7 (Reducibility). *We say that* M is reducible under Q with type A and width I, *and we write* $M \Vdash^I_Q A$, *if, for all* $\mathcal{C}$ *such that* $\mathcal{C} : L \to Q, H$, *there exist* $\mathcal{D}, V$ *such that*

1. $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$,
2. $\vDash \mathrm{width}(\mathcal{D}) \leq I$,
3. $\mathcal{D} : Q \to K$ *for some* K *such that* $V \Vdash_K A$.

Both relations, and in particular reducibility, are given in the form of unary logical relations [55]. The intuition is pretty straightforward: a term is reducible with width I if it evaluates correctly when paired with any circuit $\mathcal{C}$ which provides its free labels and if it extends $\mathcal{C}$ with a sub-circuit $\mathcal{D}$ whose width is bounded by I. Realization, on the other hand, is less immediate. For most cases, realizing type A loosely corresponds to being closed and well-typed with type A, but a value realizes an arrow type $A \multimap_{I,J} B$ when its application to a value realizing A is reducible with type B and width I.

By themselves, realization and reducibility are defined only on terms and values closed in the sense both of regular and index variables. To extend these notions to open terms and values, we adopt the standard approach of reasoning explicitly about the substitutions that would render them closed. A *closing value* *substitution* γ is a function that turns an open term M into a closed term $\gamma(M)$ by substituting a value for each free variable occurring in M. We say that γ *implements* a typing context Γ using label context Q, and we write $\gamma \vDash_Q \Gamma$, when it replaces every variable x_i in the domain of Γ with a value V_i such that $V_i \Vdash_{Q_i} \Gamma(x_i)$ and $Q = \biguplus_{x_i \in \mathrm{dom}(\Gamma)} Q_i$. A *closing index substitution* θ is similar, only it substitutes closed indices for index variables and can be applied to indices, types, contexts, values and terms alike. We say that θ implements an index context Θ, and we write $\theta \vDash \Theta$, when it replaces every index variable in Θ with a closed index term. This allows us to give the following fundamental lemma, which will be used while proving all other claims.

Lemma 1 (Core Correctness). *Let Π be a type derivation. For all $\theta \vDash \Theta$ and $\gamma \vDash_Q \theta(\Gamma)$, we have that*

$$\Pi \triangleright \Theta; \Gamma; L \vdash_c M : A; I \implies \gamma(\theta(M)) \Vdash^{\theta(I)}_{Q,L} \theta(A),$$
$$\Pi \triangleright \Theta; \Gamma; L \vdash_v V : A \implies \gamma(\theta(V)) \Vdash_{Q,L} \theta(A).$$

Proof. By induction on the size of Π, making use of Theorem 1.

Lemma 1 tells us that any well-typed term (resp. value) is reducible (resp. realizes its type) when we instantiate its free variables according to its contexts. Now that we have Lemma 1, we can proceed to proving the aforementioned results of subject reduction and total correctness. We start with the former, which unsurprisingly requires the following substitution lemmata.

Lemma 2 (Index Substitution). *Let Π be a type derivation and let I be an index such that $\Theta \vdash I$. We have that*

$$\Pi \triangleright \Theta, i; \Gamma; Q \vdash_c M : A; J \implies \Theta; \Gamma\{I/i\}; Q \vdash_c M\{I/i\} : A\{I/i\}; J\{I/i\},$$
$$\Pi \triangleright \Theta, i; \Gamma; Q \vdash_v V : A \implies \Theta; \Gamma\{I/i\}; Q \vdash_v V\{I/i\} : A\{I/i\}.$$

Proof. By induction on the size of Π.

Lemma 3 (Value Substitution). *Let Π be a type derivation and let V be a value such that $\Theta; \Phi, \Gamma_1; Q_1 \vdash_v V : A$. We have that*

$$\Pi \triangleright \Theta; \Phi, \Gamma_2, x : A; Q_2 \vdash_c M : B; I \implies \Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c M[V/x] : B; I,$$
$$\Pi \triangleright \Theta; \Phi, \Gamma_2, x : A; Q_2 \vdash_v W : B \implies \Theta; \Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_v W[V/x] : B.$$

Proof. By induction on the size of Π.

Theorem 2 (Subject Reduction). *If $Q \vdash (\mathcal{C}, M) : A; I; L$ and $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$, then $Q \vdash (\mathcal{D}, V) : A; L$.*

Proof. By induction on the derivation of $(\mathcal{C}, M) \Downarrow (\mathcal{D}, V)$ and case analysis on the last rule used in its derivation. Lemma 3 is essential to the *app,dest* and *let* cases, while Lemma 2 is used in the *fold-step* case. Lemma 1 is essential to the *box* case, as it is the only case in which the side effect of the evaluation (the circuit built by the function being boxed), whose preservation is the a matter of correctness, becomes a value (the resulting boxed circuit).

Of course, type soundness is not enough: we also want the resource analysis carried out by our type system to be correct, as stated in the following theorem.

Theorem 3 (Total Correctness). *If* $Q \vdash (\mathcal{C}, M) : A; I; L$*, then there exist* $\mathcal{D}, V$ *such that* $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$ *and* $\vDash \mathrm{width}(\mathcal{D}) \leq I$.

Proof. By definition, $Q \vdash (\mathcal{C}, M) : A; I; L$ entails that $\mathcal{C} : Q \to L, H$ and $\emptyset; \emptyset; H \vdash_c M : A; I$. Since an empty context is trivially implemented by an empty closing substitution, by Lemma 1 we get $M \Vdash^I_H A$, which by definition entails that there exist $\mathcal{D}, V$ such that $(\mathcal{C}, M) \Downarrow (\mathcal{C} :: \mathcal{D}, V)$ and $\vDash \mathrm{width}(\mathcal{D}) \leq I$.

6 A Practical Example

This section provides an example of how Proto-Quipper-R can be used to verify the resource usage of realistic quantum algorithms. In particular, we use our language to implement the QFT algorithm [11,39] and verify that the circuits it produces have width no greater than the size of their input, i.e. that the QFT algorithm does not overall use additional ancillary qubits.

```
$qft \triangleq \mathsf{fold}_j\ qftStep\ \mathsf{nil}$
$qftStep \triangleq \mathsf{lift}(\mathsf{return}\ \lambda\langle qs, q\rangle_{\mathsf{List}^j\ \mathsf{Qubit}\otimes\mathsf{Qubit}}.$
    $\mathsf{let}\ \langle n, qs\rangle = qlen\ qs\ \mathsf{in}$
    $\mathsf{let}\ revQs = rev\ qs\ \mathsf{in}$
    $\mathsf{let}\ \langle q, qs\rangle = (\mathsf{fold}_e\ (\mathsf{lift}(rotate\ n))\ \langle q, \mathsf{nil}\rangle)\ revQs\ \mathsf{in}$
    $\mathsf{let}\ q = \mathsf{apply}(\mathsf{H}, q)\ \mathsf{in}$
    $\mathsf{return}\ (\mathsf{cons}\ q\ qs))$

$rotate \triangleq \lambda n_{\mathsf{Nat}}.\mathsf{return}\ \lambda\langle\langle q, cs\rangle, c\rangle_{(\mathsf{Qubit}\otimes\mathsf{List}^e\ \mathsf{Qubit})\otimes\mathsf{Qubit}}.$
    $\mathsf{let}\ \langle m, cs\rangle = qlen\ cs\ \mathsf{in}$
    $\mathsf{let}\ rgate = \mathsf{makeRGate}\ (n + 1 - m)\ \mathsf{in}$
    $\mathsf{let}\ \langle q, c\rangle = \mathsf{apply}(rgate, \langle q, c\rangle)\ \mathsf{in}$
    $\mathsf{return}\ \langle q, \mathsf{cons}\ c\ cs\rangle$
```

Fig. 17. A Proto-Quipper-R implementation of the Quantum Fourier Transform circuit family. The usual syntactic sugar is employed.

The Proto-Quipper-R implementation of the QFT algorithm is given in Figure 17. As we walk through the various parts of the program, be aware that we will focus on the resource aspects of the algorithm, ignoring much of its actual meaning. Starting bottom-up, we assume that we have an encoding of naturals in the language and that we can perform arithmetic on them. We also assume some primitive gates and gate families: H is the boxed circuit corresponding to the Hadamard gate and has type $\mathsf{Circ}^1(\mathsf{Qubit}, \mathsf{Qubit})$, whereas the $\mathsf{makeRGate}$ function has type $\mathsf{Nat} \multimap_{0,0} \mathsf{Circ}^2(\mathsf{Qubit} \otimes \mathsf{Qubit}, \mathsf{Qubit} \otimes \mathsf{Qubit})$ and produces instances of the parametric controlled R_n gate. On the other hand, *qlen* and *rev* stand for regular language terms which implement respectively the linear list length and reverse functions. They have type $qlen : \mathsf{List}^i\,\mathsf{Qubit} \multimap_{i,0} (\mathsf{Nat} \otimes \mathsf{List}^i\,\mathsf{Qubit})$ and $rev : \mathsf{List}^i\,\mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\,\mathsf{Qubit}$ in our type system.

We now turn our attention to the actual QFT algorithm. Function *qftStep* builds a single step of the QFT circuit. The width of the circuit produced at step j is dominated by the folding of the $rotate\,n$ function, which applies controlled rotations between appropriate pairs of qubits and has type

$$(\mathsf{Qubit} \otimes \mathsf{List}^e\,\mathsf{Qubit}) \otimes \mathsf{Qubit} \multimap_{e+2,0} \mathsf{Qubit} \otimes \mathsf{List}^{e+1}\,\mathsf{Qubit}, \tag{4}$$

meaning that $rotate\,n$ rearranges the structure of its inputs, but overall does not introduce any new wire. We fold this function starting from an accumulator $\langle q, \mathsf{nil}\rangle$, meaning that we can give $\mathsf{fold}_j\ (\mathsf{lift}(rotate\,n))\ \langle q, \mathsf{nil}\rangle$ type as follows:

$$\textit{fold}\,\frac{\begin{array}{c} i,j,e;n : \mathsf{Nat};\emptyset \vdash_v \mathsf{lift}(rotate\,n) : !((\mathsf{Q} \otimes \mathsf{List}^e\,\mathsf{Q}) \otimes \mathsf{Q} \multimap_{e+2,0} \mathsf{Q} \otimes \mathsf{List}^{e+1}\,\mathsf{Q}) \\ i,j;q : \mathsf{Q};\emptyset \vdash_v \langle q, \mathsf{nil}\rangle : \mathsf{Q} \otimes \mathsf{List}^0\,\mathsf{Q} \qquad i,j \vdash j \qquad i,j \vdash \mathsf{Q} \end{array}}{i,j;n : \mathsf{Nat}, q : \mathsf{Q};\emptyset \vdash_v \mathsf{fold}_e\ \mathsf{lift}(rotate\,n)\ \langle q, \mathsf{nil}\rangle : \mathsf{List}^j\,\mathsf{Q} \multimap_{j+1,1} \mathsf{Q} \otimes \mathsf{List}^j\,\mathsf{Q}} \tag{5}$$

where Q is shorthand for Qubit and where we implicitly use the fact that $i, j \vDash \mathsf{max}(1, \mathsf{max}_{e<j}\, e + 2 + (j - 1 - e) \times 1) = j + 1$ to simplify the arrow's width annotation using *vsub* and the *arrow* subtyping rule. Next, we fold over *revQs*, which has the same elements as *qs* and thus has length j, and we obtain that the fold produces a circuit whose width is bounded by $j + 1$. Therefore, *qftStep* has type

$$!((\mathsf{List}^j\,\mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\,\mathsf{Qubit}), \tag{6}$$

which entails that when we pass it as an argument to the topmost fold together with nil we can conclude that the type of the *qft* function is

$$\textit{fold}\,\frac{\begin{array}{c} i,j;\emptyset;\emptyset \vdash_v qftStep : !((\mathsf{List}^j\,\mathsf{Qubit} \otimes \mathsf{Qubit}) \multimap_{j+1,0} \mathsf{List}^{j+1}\,\mathsf{Qubit}) \\ i;\emptyset;\emptyset \vdash_v \mathsf{nil} : \mathsf{List}^0\,\mathsf{Qubit} \qquad i \vdash i \qquad i \vdash \mathsf{Qubit} \end{array}}{i;\emptyset;\emptyset \vdash_v \mathsf{fold}_j\ qftStep\ \mathsf{nil} : \mathsf{List}^i\,\mathsf{Qubit} \multimap_{i,0} \mathsf{List}^i\,\mathsf{Qubit}} \tag{7}$$

where we once again implicitly simplify the arrow type using the fact that $i \vDash \mathsf{max}(0, \mathsf{max}_{j<i}\, j + 1 + (i - 1 - j) \times 1) = i$. This concludes our analysis and the resulting type tells us that *qft* produces a circuit of width at most i on inputs of size i, without overall using any additional wires. If we instantiate i to 3, for example, we can apply *qft* to a list of 3 qubits to obtain the circuit shown in Figure 18, whose width is exactly 3.

To conclude this section, note that for ease of exposition *qft* actually produces the *reversed* QFT circuit. This is not a problem, since the two circuits are

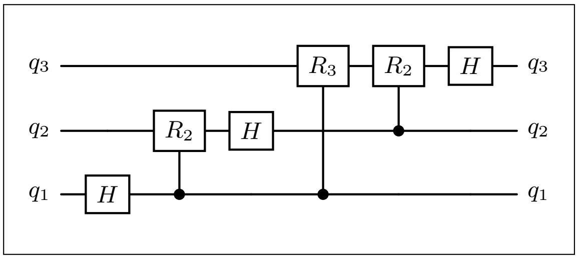


Fig. 18. The circuit of input size 3 produced by *qft* (cons q_1 cons q_2 cons q_3 nil)

equivalent resource-wise and the actual QFT circuit can be recovered by boxing the result of *qft* and reversing it via a primitive operator. Besides, note that `Quipper`'s internal implementation of the QFT is also reversed [16].

7 Related Work

The metatheory of quantum circuit description languages, and in particular of `Quipper`-style languages, has been the subject of quite some work in recent years, starting with Ross's thesis on Proto-Quipper-S [48] and going forward with Selinger and Rios's Proto-Quipper-M [46]. In the last five years, some proposals have also appeared for more expressive type systems or for language extensions that can handle non-standard language features, such as the so-called *dynamic lifting* [8,21,35], available in the `Quipper` language, or dependent types [22]. Although some embryonic contributions in the direction of analyzing the size of circuits produced using `Quipper` have been given [56], no contribution tackles the problem of deriving resource bounds *parametric* on the size of the input. In this, the ability to have types which depend on the input, certainly a feature of Proto-Quipper-D [22], is not useful for the analysis of intensional attributes of the underlying circuit, simply because such attributes are not visible in types.

If we broaden the horizon to quantum programming languages other than `Quipper`, we come across, for example, the recent works of Avanzini et al. [5] and Liu et al. [36] on adapting the classic weakest precondition technique to the cost analysis of quantum programs, which however focus on programs in an imperative language. The work of Dal Lago et al. [13] on a quantum language which characterizes complexity classes for quantum polynomial time should certainly be remembered: even though the language allows the use of higher-order functions, the manipulation of quantum data occurs directly and not through circuits. Similar considerations hold for the recent work of Hainry et al. [29] and Yamakami's algebra of functions [59] in the style of Bellantoni and Cook [6], both characterizing quantum polynomial time.

If we broaden our scope further and become interested in the analysis of the cost of classical or probabilistic programs, we face a vast literature, with contributions employing a variety of techniques on heterogeneous languages and calculi: from functional programs [2,32,33] and term rewriting systems [3,4,41] to probabilistic [34] and object-oriented programs [19,28]. In this context, the resource under analysis is often assumed to be computation *time*, which is relatively easy to analyze given its strictly monotonic nature. Circuit width, although monotonically non-decreasing, evolves in a way that depends on a non-monotonic quantity, i.e. the number of wires discarded by a circuit. As a result, width has the flavor of space and its analysis is less straightforward.

It is also worth mentioning that linear dependent types can be seen as a specialized version of refinement types [18], which have been used extensively in the literature to automatically verify interesting properties of programs [37,62]. In particular, the work of Vazou et al. on `Liquid Haskell` [57,58] has been of particular inspiration, on account of `Quipper` being embedded precisely in `Haskell`. The liquid type system [47] of `Liquid Haskell` relies on SMT solvers to discharge proof obligations and has been used fruitfully to reason about both the correctness and the resource consumption (mainly time complexity) of concrete, practical programs [30].

8 Generalization to Other Resource Types

This work focuses on estimating the *width* of the circuits produced by `Quipper` programs. This choice is dictated by the fact that the width of a circuit corresponds to the maximum number of distinct wires, and therefore individual qubits, required to execute it. Nowadays, this is considered as one of the most precious resources in quantum computing, and as such must be kept under control. However, this does not mean that our system could not be adapted to the estimation of other parameters. This section outlines how this may be possible.

First, estimating strictly monotonic resources, such as the total *number of gates* in a circuit, is possible and in fact simpler than estimating width. A *single* index term I that measures the number of gates in the circuit built by a computation would be enough to carry out this analysis. This index would be appropriately increased any time an apply instruction is executed, while sequencing two terms via let would simply add together the respective indices.

If we were instead interested in the *depth* of a circuit, then we would need a slightly different approach. Although in principle it would be possible to still rely on a single index I, this would give rise to a very coarse approximation, effectively collapsing the analysis of depth to a gate count analysis. A more precise approximation could instead be obtained by keeping track of depth *locally*. More specifically, it would be sufficient to decorate each occurrence of a wire type w with an index term I so that if a label ℓ were typed with w^I, it would mean that the sub-circuit rooted in ℓ has a depth at most equal to I.

Finally, it should be mentioned that the resources considered, i.e. the depth, width, and gate count of a circuit, can be further refined so as to take into account only *some* kinds of wires and gates. For instance, one could want to keep track of the maximum number of *qubits* needed, ignoring the number of classical bits, or at least distinguishing the two parameters, which of course have distinct levels of criticality in current quantum hardware.

9 Conclusion and Future Work

In this paper we introduced a linear dependent type system based on index refinements and effect typing for the paradigmatic calculus Proto-Quipper, with the purpose of using it to derive upper bounds on the width of the circuits produced by programs. We proved not only the classic type safety properties, but also that the upper bounds derived via the system are correct. We also showed how our system can verify a realistic quantum algorithm and elaborated on some ideas on how our technique could be adapted to other crucial resources types, like gate count and circuit depth. Ours is the first type system designed specifically for the purpose of resource analysis to target circuit description languages such as `Quipper`. Technically, the main novelties are the smooth combination of effect typing and index refinements, but also the proof of correctness, in which reducibility and effects are shown to play well together.

Among topics for further work, we can identify three main research directions. First and foremost, it would be valuable to investigate the ideas presented in this paper from a more practical perspective, that is, to provide a prototype implementation of the language with its type-checking procedure. The necessity to count the wires present in the context (e.g. when typing abstractions) makes it difficult to embed Proto-Quipper-R into existing languages, even those that, in principle, seem like ideal hosts, like `Liquid Haskell` [57] or `Granule` [42]. Because of this, we think that it would be better to produce a standalone implementation of Proto-Quipper-R that interfaces directly with SMT solvers to discharge the semantic judgments that are used pervasively in the typing rules.

Staying instead on the theoretical side of things, on one hand we have the prospect of denotational semantics: most incarnations of Proto-Quipper are endowed with categorical semantics that model both circuits and the terms of the language that build them [21,22,35,46]. We already mentioned how the intensional nature of the quantity under analysis renders the formulation of an abstract categorical semantics for Proto-Quipper-R and its circuits a nontrivial task, but we believe that one such semantics would help Proto-Quipper-R fit better in the Proto-Quipper landscape.

On the other hand, in Section 8 we briefly discussed how our system could be modified to handle the analysis of different resource types. It would be interesting to test this path and to investigate the possibility of *actually generalizing* our resource analysis, that is, of making it parametric on the kind of resource being analyzed. This would allow for the same program in the same language to be amenable to different forms of verification, in a very flexible fashion.

Acknowledgments The research leading to these results has received funding from the European Union - NextGenerationEU through the Italian Ministry of University and Research under PNRR - M4C2 - I1.4 Project CN00000013 "National Centre for HPC, Big Data and Quantum Computing".

References

1. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: Proc. of LICS '05 (2005)
2. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the complexity of functional programs: Higher-order meets first-order. In: Proc. of ICFP 2015 (2015)
3. Avanzini, M., Moser, G.: Complexity analysis by rewriting. In: Proc. of FLOPS 2008 (2008)
4. Avanzini, M., Moser, G.: Tyrolean Complexity Tool: Features and Usage. In: Proc. of RTA 2013. vol. 21 (2013)
5. Avanzini, M., Moser, G., Pechoux, R., Perdrix, S., Zamdzhiev, V.: Quantum expectation transformers for cost analysis. In: Proc. of LICS '22 (2022)
6. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions (extended abstract). In: STOC '92 (1992)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press (2021)
8. Colledan, A., Dal Lago, U.: On Dynamic Lifting and Effect Typing in Circuit Description Languages. In: TYPES 2022. vol. 269 (2023)
9. Collins, H., Nay, C.: Ibm unveils 400 qubit-plus quantum processor and next-generation ibm quantum system two (2022), https://is.gd/WPV7lO, retrieved on Oct. 15, 2023
10. Conover, E.: Light-based quantum computer jiuzhang achieves quantum supremacy (2020), https://is.gd/kFv6IOy, retrieved on Oct. 15, 2023
11. Coppersmith, D.: An approximate fourier transform useful in quantum factoring, ibm research report rc19642 (2002)
12. Dal Lago, U., Gaboardi, M.: Linear dependent types and relative completeness. In: Proc. of LICS '11 (2011)
13. Dal Lago, U., Masini, A., Zorzi, M.: Quantum implicit computational complexity. TCS **411**(2) (2010)
14. Dal lago, U., Petit, B.: Linear dependent types in a call-by-value scenario. In: Proc. of PPDP '12 (2012)
15. Dal lago, U., Petit, B.: The geometry of types. In: Proc. of POPL '13 (2013)
16. Eisenberg, R., Green, A., Lumsdaine, P., Kim, K., Mau, S.C., Mohan, B., Ng, W., Ravelomanantsoa-Ratsimihah, J., Ross, N., Scherer, A., Selinger, P., Valiron, B., Virodov, A., Zdancewic, S.: Quipper.libraries.qft, https://is.gd/AEJmp9, retrieved on Oct. 15, 2023
17. Fingerhuth, M., Babej, T., Wittek, P.: Open source software in quantum computing. PLOS ONE **13**(12) (2018)
18. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proc. of PLDI '91 (1991)
19. Frohn, F., Giesl, J.: Complexity analysis for java with aprove. In: Proc. of IFM 2017 (2017)
20. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In: Proc. of RC (2020)
21. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-quipper with dynamic lifting. In: Proc. of POPL '23 (2023)
22. Fu, P., Kishida, K., Selinger, P.: Linear dependent type theory for quantum programming languages: Extended abstract. In: Proc. of LICS '20 (2020)
23. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Proc. of POPL '13 (2013)

24. Gay, S.J.: Quantum programming languages: Survey and bibliography. Math. Struct. Comput. Sci. **16**(4) (2006)
25. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Proc. of RC (2013)
26. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper. In: Proc. of PLDI (2013)
27. Grover, L.K.: A fast quantum mechanical algorithm for database search (1996)
28. Hainry, E., Péchoux, R.: Type-based heap and stack space analysis in Java (2013), technical report
29. Hainry, E., Péchoux, R., Silva, M.: A programming language characterizing quantum polynomial time. In: Proc. of FoSSaCS 2023 (2023)
30. Handley, M., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in liquid haskell. PACMPL **4** (2019)
31. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Phys. Rev. Lett. **103** (2009)
32. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ml. In: Computer Aided Verification (2012)
33. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Programming Languages and Systems (2010)
34. Kaminski, B.L., Katoen, J.P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. In: Programming Languages and Systems. Berlin, Heidelberg (2016)
35. Lee, D., Perrelle, V., Valiron, B., Xu, Z.: Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In: Proc. of FSTTCS (2021)
36. Liu, J., Zhou, L., Barthe, G., Ying, M.: Quantum weakest preconditions for reasoning about expected runtimes of quantum programs. In: Proc. of LICS '22 (2022)
37. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proc. of ICFP '03 (2003)
38. Martinis, J.: Quantum supremacy using a programmable superconducting processor (2019), https://is.gd/v3VXFi, retrieved on Oct. 15, 2023
39. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press (2010)
40. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design: Recent Insights and Advances. Springer Berlin Heidelberg (1999)
41. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. Journal of Automated Reasoning **51**(1) (2013)
42. Orchard, D., Liepelt, V.B., Eades III, H.: Quantitative program reasoning with graded modal types. In: Proc. of ICFP '19. vol. 3 (2019)
43. Palsberg, J.: Toward a universal quantum programming language. XRDS: Crossroads **26**(1) (2019)
44. Paykin, J., Rand, R., Zdancewic, S.: Qwire: A core language for quantum circuits. In: Proc. of POPL '17 (2017)
45. Preskill, J.: Quantum computing and the entanglement frontier (2012)
46. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. In: Proc. of QPL '17. vol. 266 (2017)
47. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proc. of PLDI '08 (2008)
48. Ross, N.: Algebraic and Logical Methods in Quantum Computation. Ph.D. thesis, Dalhousie University (2015)
49. Sanders, J.W., Zuliani, P.: Quantum programming. In: Proc. of MPC 2000 (2000)
50. Schlosshauer, M.: Decoherence and the Quantum-To-Classical Transition. Springer Berlin Heidelberg (2007)

51. Selinger, P.: A brief survey of quantum programming languages. In: Proc. of FLOPS 2004 (2004)
52. Selinger, P.: Towards a quantum programming language. Math. Struct. Comput. Sci. **14**(4) (2004)
53. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. In: Proc. of TLCA (2005)
54. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proc. of FOCS '94 (1994)
55. Skorstengaard, L.: An introduction to logical relations (2019)
56. Valiron, B.: Automated, parametric gate count of quantum programs (2016), https://is.gd/XIm3lh, retrieved on Oct. 15 2023
57. Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: Experience with refinement types in the real world. In: Proc. of Haskell '14 (2014)
58. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for Haskell. In: Proc. of ICFP '14 (2014)
59. Yamakami, T.: A schematic definition of quantum polynomial time computability. The Journal of Symbolic Logic **85**(4) (2020)
60. Yanofsky, N.S., Mannucci, M.A.: Quantum Computing for Computer Scientists. Cambridge University Press, USA, 1st edn. (2008)
61. Ying, M.: Foundations of Quantum Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2016)
62. Çiçek, E., Garg, D., Acar, U.: Refinement Types for Incremental Computational Complexity. In: Programming Languages and Systems, vol. 9032. Springer Berlin Heidelberg (2015)
63. Ömer, B.: Classical concepts in quantum programming. Int. J. Theor. Phys. **44**(7) (2005)

On the Hardness of Analyzing Quantum Programs Quantitatively

Martin Avanzini[1], Georg Moser[2], Romain Péchoux[3](✉), and Simon Perdrix[3]

[1] Centre Inria d'Université Côte d'Azur, Valbonne, France
martin.avanzini@inria.fr
[2] Universität Innsbruck, Innsbruck, Austria
georg.moser@uibk.ac.at
[3] CNRS-Inria-Université de Lorraine, LORIA, Nancy, France
{romain.pechoux,simon.perdrix}@loria.fr

Abstract. In this paper, we study quantitative properties of quantum programs. Properties of interest include (positive) almost-sure termination, expected runtime or expected cost, that is, for example, the expected number of applications of a given quantum gate, etc. After studying the completeness of these problems in the arithmetical hierarchy over the Clifford+T fragment of quantum mechanics, we express these problems using a variation of a quantum pre-expectation transformer, a weakest pre-condition based technique that allows to symbolically compute these quantitative properties. Under a smooth restriction—a restriction to polynomials of bounded degree over a real closed field—we show that the quantitative problem, which consists in finding an upper-bound to the pre-expectation, can be decided in time double-exponential in the size of a program, thus providing, despite its great complexity, one of the first decidable results on the analysis and verification of quantum programs. Finally, we sketch how the latter can be transformed into an efficient synthesis method.

1 Introduction

Motivations. Quantum computation is a promising and emerging computational paradigm which can efficiently solve problems considered to be intractable on classical computers [41,20]. However, the unintuitive nature of quantum mechanics poses challenging questions for the design and analysis of corresponding quantum programming. Indeed, the quantum program dynamics are considerably more complicated compared to the behavior of classical or probabilistic programs. Therefore, formal reasoning requires the development of novel methods and tools, a development that has already started and recently gathered momentum in various areas, like *design automation* [43,22], *programming languages* [39,2,31,23,15], *verification* [36,11], etc.

Among these formal methods, those that allow us to obtain quantitative properties on quantum programs are particularly interesting. They can be used to obtain relevant information about the computations of a quantum program,

```
RUS ≜ i^N = 0;
      x^B = tt;
      while x do {
        q2 = |0⟩ ;
        q2 *= H;
        q2 *= T;
        i = i + 1
        q2, q1 *= CNOT;
        q2 *= H;
        q2, q1 *= CNOT;
        q2 *= T;
        i = i + 1
        q2 *= H;
        x = meas q2
      }
.
```

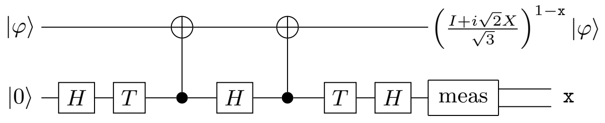

$|\varphi\rangle$
$|0\rangle$
H
T
H
T
H
meas
$\left(\frac{I+i\sqrt{2}X}{\sqrt{3}}\right)^{1-x}|\varphi\rangle$
x

Fig. 1. Repeat-until-success program RUS and step-circuit.

such as the number of qubits used and the number of unitary operators used, thus enabling the corresponding compiled quantum circuit to be optimized (for example, by minimizing the use of gates that are hard to make fault-tolerant, or by reducing the number of qubits) or to avoid undesirable behavior such as non-termination. Another quantitative property of interest may also be the question whether or not a program *terminates almost-surely*, that is, whether its probability of non-termination is zero or not. Similarly, we could aim to capture the *expected values* of (classical) program variables upon program termination. The latter can also be employed to reason about the *expected runtime* or the *expected cost* of quantum programs, if we suitably instrument the code with counter variables.

To illustrate this, the program of Figure 1 implements a Repeat-Until-Success algorithm that can be used to simulate quantum unitary operators on input qubit q_1 by using repeated measurements. The quantum step-circuit on the right part corresponds to one iteration of the loop. Variable i in the program just acts as a counter for T-gates. Hence an analysis on the expected value of variable i can be used to infer an upper-bound on the expected *T-count*, i.e., the expected number of times a T-gate is used in the fully compiled quantum circuit. Such an approach offers the advantage to allow the programmer to implement quantum programs using fewer T-gates, which are costly to implement fault-tolerantly [10,16], and it therefore provides a simple quantum program to illustrate that the study of quantitative properties is paramount.

In [6,30], new methodologies named *quantum expectation transformers* based on *predicate transformers* [13,28] and *expectation transformers* [32,17] have been put forward to naturally express and study the quantitative properties of quantum programs. However, no attempt was made to automate the corresponding techniques or delineate how complicated such an *automation* could be. Automation of these formal verification techniques in the context of quantum programs is a particularly difficult problem. Indeed, the consideration of Hilbert spaces as a mathematical framework for describing principles and laws of quantum mechanics makes it seemingly impossible to reason fully automatically about quantitative properties of quantum program: they involve computational objects of exponential dimensions (in the number of qubits) with scalars ranging over an uncountable domain (i.e., complex numbers $\mathbb{C}$). This problem is directly linked to the fact that the set $\mathbb{C}$ includes non-computable numbers [42] and that testing the inequality $\leq$ or the equality $=$ of two real numbers is not decidable, even if one restricts their study to computable real numbers. Consequently, the particular nature of quantum programs and of their semantic domain, Hilbert spaces, makes it impossible to directly apply the results obtained in the classical and probabilistic setting [37,24].

Contributions. In this paper, we study the hardness of the quantitative properties of mixed classical-quantum programs and provide a first step towards their (full) automation using quantum expectation transformers.

To this end, we restrict the considered quantum gates to the *Clifford+T fragment*, which is known to be the simplest approximately universal fragment of quantum mechanics [1]. Clifford+T makes it possible to only consider quantum states with algebraic amplitudes, thus restricting the study to a countable domain. It implies that our results can accommodate quantum gates employed in actual hardware, recently employed to claim *quantum advantage*, cf [3]. Moreover, the obtained results are very general as it can be extended to any set of gates with algebraic coefficients.

As motivated, our first contribution is about the general hardness of deciding quantitative properties for mixed classical-quantum programs. For a given input state, we study properties such as *(positive) almost-sure termination*, (P)AST for short; *testing problems*, $\text{TEST}_{\mathcal{R}}$, which consist in comparing a quantum expectation (for example, the mean value of a variable) with a given value (an algebraic and positive real number) wrt the relation $\mathcal{R}$; and the *finiteness problem*, $\text{TEST}_{\neq\infty}$, which consists in checking that a quantum expectation is finite. For each of those problems, we also study the related *universal problem*, which consists in checking the corresponding property for every input. We establish a precise mapping (Theorem 1) of the inherent complexity of each problem in the arithmetical hierarchy [34] that is summarized in Table 1 (provided in Section 3). E.g., AST is Π^0_2-complete while PAST is Σ^0_2-complete.

Our second contribution aims to overcome the aforementioned undecidability results. For that, we study approximations. More precisely, we focus on *inferring* bounding functions (in general depending on the input) on the expected values of classical program variables upon termination. The decision problem has thus been altered to an inference problem. Further, we restrict the set of potential bounding functions. As a suitable class of functions, we consider polynomials over the real-closed field of the algebraic numbers. The restriction to algebraic numbers guarantees that comparison operations between real numbers remain decidable. On the other hand, for any real closed field, quantifier elimination for formulas over polynomials is decidable, that is, there exists a double-exponential algorithm computing a quantifier-free formula equivalent to the original formula [21]. This recasting of the problem and restriction of the solution space suffices to render the problem decidable. The inference algorithm established remains double-exponential (Theorem 4), thus of similar complexity as the underlying quantifier elimination procedure.

Finally, our last contribution (Section 5) studies effective automation of the inference of upper bounds on the expected values of program variables. To improve upon the double-exponential complexity, we further restrict the class of polynomials considered, that is, to degree-2 polynomials and sketch how techniques from optimization theory can be employed. Several simple quantum algorithms such as program `RUS` can be analyzed using this approach (Example 6). This further reduction in expressivity allows the encoding of the problem in SMT and thus paves the way towards (full) automation.

Related Work. Predicate transformers [13,28]—on which our work is based—were introduced as a method for reasoning about the semantics of imperative programs. They have been adapted to the probabilistic setting, leading to the notion of expectation transformer [32,17], which has been used to reason about expected values [26,8], runtimes [27,33], and costs [7,4,33], and to the quantum paradigm, leading to the notion of quantum pre-expectation transformer [35,30,6].

The problem of studying the difficulty of analyzing quantitative program properties has been deeply studied in the classical setting. To mention a few, [14] and [37] study termination properties and runtime/derivational properties of first-order programs, respectively. Further, in [24] completeness results for various quantitative properties of (pure) probabilistic programs have been established. The inference problem of expectation transformers, i.e., establishing an implementation that automates the search for pre-expectations, has been studied extensively. Examples of successful implementation are presented in [33,7,8]. Up to now, however, no practical, feasible studies have been carried out on quantum languages. Among the techniques using quantum expectation transformers, we believe [6] to be the most amenable to automation. Indeed, by lifting *upper invariants* of [27] to the quantum setting, it enables approximate reasoning and eliminate the need to reason about fixpoints or limits, stemming from the semantics of loops.

2 Quantum Programming Language

In this section, we introduce the syntax and operational semantics of the considered mixed-quantum imperative programming language.

Syntax. We make use of three basic datatypes $\mathcal{B}$, $\mathcal{N}$ and $\mathcal{Q}$ for Boolean, numbers (non-negative integers), and qubit data, respectively. Let $\mathcal{K}$ be an arbitrary

$$
\begin{array}{rl}
\mathcal{N}\texttt{Exp} \ni \texttt{n}, \texttt{n}_1, \texttt{n}_2 & ::= \texttt{x}^{\mathcal{N}} \mid n \in \mathbb{N} \mid \texttt{n}_1 + \texttt{n}_2 \mid \texttt{n}_1 - \texttt{n}_2 \mid \texttt{n}_1 \times \texttt{n}_2 \\
\mathcal{B}\texttt{Exp} \ni \texttt{b}, \texttt{b}_1, \texttt{b}_2 & ::= \texttt{x}^{\mathcal{B}} \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{n}_1 = \texttt{n}_2 \mid \texttt{n}_1 < \texttt{n}_2 \mid \neg\texttt{b} \mid \texttt{b}_1 \wedge \texttt{b}_2 \mid \texttt{b}_1 \vee \texttt{b}_2 \\
\texttt{Exp} \ni \texttt{e}, \texttt{e}_1, \texttt{e}_2 & ::= \texttt{n} \mid \texttt{b} \\
\texttt{Stmt} \ni \texttt{stm}, \texttt{stm}_1, \texttt{stm}_2 & ::= \texttt{skip} \mid \texttt{x}^{\mathcal{K}} = \texttt{e}^{\mathcal{K}} \mid \texttt{stm}_1\texttt{;}\ \texttt{stm}_2 \mid \texttt{if}\ \texttt{b}^{\mathcal{B}}\ \texttt{then}\ \texttt{stm}_1\ \texttt{else}\ \texttt{stm}_2 \\
& \quad \mid \texttt{while}\ \texttt{b}^{\mathcal{B}}\ \texttt{do}\ \texttt{stm} \mid \overline{\texttt{q}}^{\mathcal{Q}} \mathrel{*}= \texttt{U} \mid \texttt{x}^{\mathcal{B}} = \texttt{meas}\ \texttt{q}^{\mathcal{Q}}
\end{array}
$$

Fig. 2. Syntax of quantum programs.

classical type in $\{\mathcal{B}, \mathcal{N}\}$. Each program variable comes with a fixed datatype and can be optionally annotated by its type as a superscript. In what follows, we will use $\texttt{x}, \texttt{x}', \texttt{y}, \ldots$ to denote classical variables of type $\mathcal{K}$ and $\texttt{q}, \texttt{q}', \ldots$ to denote quantum variables of type $\mathcal{Q}$. A program, denoted $\texttt{P}$, is simply a statement; see Figure 2. Program statements are either classical assignments, conditionals, sequences, loops, *quantum assignments* $\overline{\texttt{q}}^{\mathcal{Q}} \mathrel{*}= \texttt{U}$, or *measurements* $\texttt{x}^{\mathcal{B}} = \texttt{meas}\ \texttt{q}^{\mathcal{Q}}$. A quantum assignment consists in the application of a quantum unitary gate $\texttt{U}$ of arity $ar(\texttt{U})$ to a sequence of qubits $\overline{\texttt{q}} \triangleq \texttt{q}_1, \ldots, \texttt{q}_{ar(\texttt{U})}$. As we will see in the semantics section, a unitary matrix U will be associated with each quantum gate $\texttt{U}$. A measurement performs a single qubit measurement of $\texttt{q}$ in the computational basis: the outcome is a Boolean value and the quantum state evolves accordingly. For a given syntactic construct t, let $\mathcal{B}(t)$ (respectively $\mathcal{N}(t)$, $\mathcal{Q}(t)$) be the set of Boolean (respectively, number, qubit) variables in t.

Notice that the language encompasses qubit-initializing in the basis states. In particular, we will use $\texttt{q}^{\mathcal{Q}} = |0\rangle$ as syntactic sugar for $\texttt{x} = \texttt{meas q; if x then q} \mathrel{*}= \texttt{X else skip}$, for $\texttt{X}$ being the Pauli X gate and for some fresh variable $\texttt{x}$ of type $\mathcal{B}$.

Example 1. Consider the program of Figure 3, adapted from [6], as a simple leading example. Let H be the unitary operator computing the Hadamard gate. This program simulates coin tossing by repeatedly measuring the qubit $\texttt{q}$, until the measurement outcome $\texttt{ff}$ occurs. The probability to terminate within n steps depends on the initial state $\rho = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$ (a density matrix in $\mathbb{C}^{2\times 2}$, which implies $\alpha + \delta = 1$ and $\gamma = \bar{\beta}$) of the qubit $\texttt{q}$. Variable $\texttt{i}$ is increased by one at each iteration, and hence, when the program terminates, $\texttt{i}$ stores as final value the number of loop iterations performed. The overall probability of termination is 1. The mean value of variable $\texttt{i}$, that is, the expected number of loop iterations, depends on the program input, in particular on the initial quantum state. After termination, for an initial state $\rho = \begin{pmatrix} \alpha & \beta \\ \bar{\beta} & \delta \end{pmatrix}$, its expected value is given by

$$F(\rho) = p_0 \times 1 + \sum_{i=1}^{\infty} \frac{p_1}{2^i}(i+1) = p_0 + p_1 + 2p_1 = 1 + (\alpha - \beta - \bar{\beta} + \delta) = 2 - (\beta + \bar{\beta}),$$

where $p_0 = \frac{\alpha+\beta+\bar{\beta}+\delta}{2} = \frac{1+\beta+\bar{\beta}}{2}$ and $p_1 = 1 - p_0$ are the probabilities of measuring $|0\rangle$ and $|1\rangle$, respectively, on the first iteration of the loop. For instance, for a qubit

```
Cntoss  ≜  x^B = tt;
           i^N = 0;
           while x do {
             i = i + 1;  ⎫
             q^Q *= H;   ⎬ ≜ stm
             x = meas q  ⎭
           }
```

with $H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Fig. 3. Quantum Coin tossing

initialized in state $|\phi\rangle = \sqrt{1/3}\,|0\rangle + \sqrt{2/3}\,|1\rangle$, the corresponding density matrix is $\rho_{|\phi\rangle} = |\phi\rangle\langle\phi| = \begin{pmatrix} 1/3 & \sqrt{2}/3 \\ \sqrt{2}/3 & 2/3 \end{pmatrix}$ and hence the expected number of loop iterations is $F(\rho_{|\phi\rangle}) = 2 - 2\sqrt{2}/3$. It will be simply 2 in the case of an initialization in the computational basis $|\phi\rangle = |0\rangle$ or $|\phi\rangle = |1\rangle$.

Operational Semantics. Following [6], we model the dynamics of our language as a probabilistic abstract reduction system (see [9]), a transition system where reduction is defined as a relation over probability distributions.

Probabilistic abstract reduction systems. Given a subset $\mathbb{K}$ of $\mathbb{R}$, let $\mathbb{K}^+$ be the set of non-negative numbers in $\mathbb{K}$, i.e., $\mathbb{K}^+ \triangleq \mathbb{K} \cap \{x \mid x \geq 0\}$ and let $\mathbb{K}^\infty$ be defined by $\mathbb{K}^\infty \triangleq \mathbb{K} \cup \{\infty\}$.

A discrete (sub)*distribution* δ over a set A is a function $\delta : A \to [0,1]$ with countable support $\mathrm{supp}(\delta) \triangleq \{a \in A \mid \delta(a) \neq 0\}$ that maps an element a of A to a probability $\delta(a)$ such that $|\delta| \triangleq \sum_{a \in \mathrm{supp}(\delta)} \delta(a) = 1$ ($|\delta| \leq 1$). Any (sub)distribution δ can be written as $\{\delta(a) : a\}_{a \in \mathrm{supp}(\delta)}$. The set of subdistributions over A, denoted by $\mathcal{D}(A)$, is closed under denumerable convex combinations $\sum_i p_i \cdot \delta_i \triangleq \lambda a. \sum_i p_i \delta_i(a)$, with $p_i \in [0,1]$ and $\sum_i p_i \leq 1$. Slightly simplifying standard notation, given $f : A \to \mathbb{R}^{+\infty}$ and a subdistribution $\delta \in \mathcal{D}(A)$, we define $\mathbb{E}_\delta(f)$, *the expectation of* f *on* δ, by $\mathbb{E}_\delta(f) \triangleq \Sigma_{a \in supp(\delta)} \delta(a) f(a)$. Note that $\mathbb{E}_\delta(f) \in \mathbb{R}^{+\infty}$ is always defined, since the images of f are non-negative reals.

Bournez and Garnier [9] introduced the notion of *Probabilistic Abstract Reduction System* (PARS) as a means to study reduction systems that evolve probabilistically. A PARS $\to$ on A is a binary relation $\cdot \to \cdot \subseteq A \times \mathcal{D}(A)$. The intended meaning is that when $a \to \delta$, then a reduces to $b \in \mathrm{supp}(\delta)$ with probability $\delta(b)$. Here, we focus on *deterministic* PARSs, i.e., PARSs $\to$ with $a \to \delta_1$ and $a \to \delta_2$ implies $\delta_1 = \delta_2$. An object $a \in A$ is called *terminal* if there is no rule $a \to \delta$, which we write as $a \not\to$.

Every deterministic PARS $\to$ over A naturally lifts to a reduction relation $\twoheadrightarrow$ over distributions so that $\delta \twoheadrightarrow \varepsilon$, if the reduct distribution ε is obtained from δ by replacing reducts in $\mathrm{supp}(\delta)$ according to the PARS $\to$. In fact, we define this lifting in terms of a ternary relation $\cdot \overset{\cdot}{\twoheadrightarrow} \cdot \subseteq \mathcal{D}(A) \times \mathbb{R}^+ \times \mathcal{D}(A)$ on distributions, where in a step $\delta \overset{c}{\twoheadrightarrow} \varepsilon$ the *weight* c signifies the probability that a reduction has occurred. This relation is defined wrt. the following three rules.

$$\frac{a \not\to}{\{1 : a\} \overset{0}{\twoheadrightarrow} \{1 : a\}} \qquad \frac{a \to \delta}{\{1 : a\} \overset{1}{\twoheadrightarrow} \delta} \qquad \frac{\delta_i \overset{c_i}{\twoheadrightarrow} \epsilon_i \quad \sum_i p_i \leq 1}{\sum_i p_i \cdot \delta_i \xrightarrow{\sum_i p_i c_i} \sum_i p_i \cdot \epsilon_i}$$

We may sometimes use the n-fold ($n \geq 0$) composition of $\overset{\cdot}{\twoheadrightarrow}$, denoted $\overset{\cdot}{\twoheadrightarrow}^n$, given by $\delta \overset{c}{\twoheadrightarrow}^n \epsilon$ if $\delta \overset{c_1}{\twoheadrightarrow} \cdots \overset{c_n}{\twoheadrightarrow} \epsilon$ and the weights satisfy $c = \sum_{i=1}^n c_i$. Notice that since $\to$ is deterministic, so is $\overset{c}{\twoheadrightarrow}$, in the sense that $\delta \overset{c_1}{\twoheadrightarrow} \epsilon_1$ and $\delta \overset{c_2}{\twoheadrightarrow} \epsilon_2$ implies $c_1 = c_2$ and $\epsilon_1 = \epsilon_2$. Thus, in particular, for every $a \in A$ there is precisely one (infinite) reduction

$$\{1 : a\} = \delta_0 \overset{c_0}{\twoheadrightarrow} \delta_1 \overset{c_1}{\twoheadrightarrow} \delta_2 \overset{c_2}{\twoheadrightarrow} \delta_3 \twoheadrightarrow \cdots .$$

For any $b \in A$, $\delta_i(b)$ gives the probability that a reduces to b in i steps. Note that when b is terminal, this probability only increases along reductions (i.e., $\delta_i(b) \leq \delta_{i+1}(b)$ for all i). This justifies that we define the *terminal distribution* of a as the distribution $\delta(b) \triangleq \lim_{i\to\infty} \delta_i(b)$. Note that $\delta(b)$ gives the probability that a reaches b in an arbitrary (but finite) number of steps. Since the weights c_i indicate the probability that a step has been performed from δ_i to δ_{i+1}, the infinite sum $\sum_{i=0}^{\infty} c_i \in \mathbb{R}^{+\infty}$ gives the expected number of reduction steps carried out, the *expected derivation length of a* [5].

For a PARS $\to$, we denote by $\mathsf{term}_\to : A \to \mathcal{D}(A)$ the function associating with each $a \in A$ its terminal distribution. The *expected derivation length function* $\mathsf{edl}_\to : A \to \mathbb{R}^{+\infty}$ associates each $a \in A$ to its expected derivation length. The PARS $\to$ is *almost surely terminating* [40] (*a.s. terminating* for short) if $a \in A$ reduces to a terminal object $b \not\to$ with probability 1, that is, if $|\mathsf{term}_\to(a)| = 1$ for every a. It is *positive almost surely terminating*, if the expected derivation length is always finite, that is, $\mathsf{edl}_\to(a) < \infty$ for all $a \in A$.

Apart from termination, we are interested also in questions related to functional correctness, such as (i) what is the probability that a reaches a terminal b, (ii) what is the probability that a reaches a terminal satisfying predicate P, and more generally, (iii) which value does a function $f : A \to \mathbb{R}^{+\infty}$ take, in expectation, when fully reducing an object a. In the literature [32], one tool to answer all of these are given by *weakest pre-expectation transformers*, the natural generalization of classical weakest pre-condition transformers to a quantitative, probabilistic setting. We suite this notion to PARSs.

Definition 1 (Weakest pre-expectation). *The* weakest pre-expectation *for a PARS $\to$ over A is given by the function*

$$\mathsf{wp}_\to : (A \to \mathbb{R}^{+\infty}) \to (A \to \mathbb{R}^{+\infty})$$
$$\mathsf{wp}_\to \triangleq \lambda f.\lambda a.\ \mathbb{E}_{\mathsf{term}_\to(a)}(f).$$

For $\mathbf{1}_b$ the indicator function evaluating to 1 on argument b and to 0 otherwise, and by seeing a predicate P as a $0,1$-valued function, $\mathsf{wp}_\to\ \mathbf{1}_b\ a$ answers

$$\frac{}{(\texttt{skip}, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\downarrow, s, \rho)\}}\ (\text{Skip}) \qquad \frac{}{(\texttt{x} = \texttt{e}, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\downarrow, s[\texttt{x} := [\![\texttt{e}]\!]^s], \rho)\}}\ (\text{Exp})$$

$$\frac{}{(\overline{\texttt{q}} \mathrel{*}= \texttt{U}, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\downarrow, s, \Phi_{U_{\overline{\texttt{q}}}}(\rho))\}}\ (\text{Op})$$

$$\frac{}{(\texttt{x} = \texttt{meas}\ \texttt{q}_i, s, \rho) \rightarrow_{\mathrm{Q}} \{tr(M_{k,i}\rho) : (\downarrow, s[\texttt{x} := k], m_{k,i}(\rho))\}_{k \in \{0,1\}}}\ (\text{Meas})$$

$$\frac{(\texttt{stm}_1, s, \rho) \rightarrow_{\mathrm{Q}} \{p_i : (\texttt{stm}^i_{\downarrow}, s^i, \rho^i)\}_{i \in I}}{(\texttt{stm}_1\texttt{;}\ \texttt{stm}_2, s, \rho) \rightarrow_{\mathrm{Q}} \{p_i : (\texttt{stm}^i_{\downarrow}\texttt{;}\ \texttt{stm}_2, s^i, \rho^i)\}_{i \in I}}\ (\text{Seq})$$

$$\frac{[\![\texttt{b}]\!]^s \in \{0,1\}}{(\texttt{if b then stm}_1\ \texttt{else stm}_0, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\texttt{stm}_{[\![\texttt{b}]\!]^s}, s, \rho)\}}\ (\text{Cond})$$

$$\frac{[\![\texttt{b}]\!]^s = 0}{(\texttt{while b do stm}, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\downarrow, s, \rho)\}}\ (\text{Wh}_0)$$

$$\frac{[\![\texttt{b}]\!]^s = 1}{(\texttt{while b do stm}, s, \rho) \rightarrow_{\mathrm{Q}} \{1 : (\texttt{stm; while b do stm}, s, \rho)\}}\ (\text{Wh}_1)$$

Fig. 4. Operational semantics in terms of PARS.

question (i), $\mathsf{wp}_{\rightarrow}\ P\ a$ answers (ii), and generally $\mathsf{wp}_{\rightarrow}\ f\ a$ answers question (iii). Note also that a PARS is a.s. terminating iff $\mathsf{wp}_{\rightarrow}\ (\lambda b.\,1)\ a = 1$ for each $a \in A$. On the other hand, positive a.s. termination cannot be expressed through an application of $\mathsf{wp}_{\rightarrow}$.

Quantum programs as PARSs. We now endow quantum programs with an operational semantics defined in terms of a PARS. Given a totally ordered set of qubits $Q = \{\texttt{q}_1, \dots, \texttt{q}_n\}$, let $\mathcal{H}_Q$ be the 2^n-dimensional Hilbert space defined by $\mathcal{H}_Q \triangleq \otimes_{i=1}^n \mathcal{H}_{\texttt{q}_i}$, with $\mathcal{H}_{\texttt{q}} = \mathbb{C}^2$ being the vector space of computational basis $\{|0\rangle, |1\rangle\}$ and $\otimes$ being the tensor product. With $\langle k|$ we denote the transpose conjugate of $|k\rangle$, for $k \in \{0,1\}$. Let $\mathcal{M}(\mathcal{H}_Q)$ be the set of complex square matrices acting on the Hilbert space $\mathcal{H}_Q$, i.e., $\mathcal{M}(\mathcal{H}_Q) = \mathbb{C}^{2^n \times 2^n}$. Given $M \in \mathcal{M}(\mathcal{H}_Q)$, $M^\dagger$ denotes the transpose conjugate of M, and I_{2^n} denotes the identity matrix over $\mathcal{M}(\mathcal{H}_Q)$. We will write I when the dimension is clear from the context.

Let $\mathfrak{D}(\mathcal{H}_Q) \subsetneq \mathcal{M}(\mathcal{H}_Q)$ be the set of all *density operators* (or quantum states), i.e., positive semi-definite matrices of trace equal to 1 on $\mathcal{H}_Q$. Density operators can be viewed as the mathematical representation of a (mixed) *quantum state.* A *unitary operator* U is a matrix in $\mathcal{M}(\mathcal{H}_Q)$ such that $UU^\dagger = U^\dagger U = I$. A *superoperator* $\Phi_U : \mathfrak{D}(\mathcal{H}_Q) \to \mathfrak{D}(\mathcal{H}_Q)$, an endomorphism over density operators, is attached to each unitary operator U and defined by $\Phi_U \triangleq \lambda\rho.U\rho U^\dagger$. By definition, Φ_U is a completely positive trace preserving linear map. Indeed, $tr(U\rho U^\dagger) = tr(\rho)$, by unitarity. Hence $U\rho U^\dagger$ is a density operator in $\mathfrak{D}(\mathcal{H}_Q)$ for each $\rho \in \mathfrak{D}(\mathcal{H}_Q)$.

Regarding measurements, for each i, $1 \leq i \leq \mathit{card}(Q)$, we define $M_{k,i} \in \mathcal{M}(\mathcal{H}_Q)$, with $k \in \{0,1\}$, by $M_{0,i} \triangleq I_{2^{i-1}} \otimes (|0\rangle \langle 0|) \otimes I_{2^{n-i}}$ and $M_{1,i} \triangleq I - M_{0,i}$. The measurement of the qubit q_i (in the computational basis) of a density matrix $\rho \in \mathfrak{D}(\mathcal{H}_Q)$, produces the classical outcome $k \in \{0,1\}$ with probability $tr(M_{k,i}\rho)$. The (normalized) quantum state, after the measurement, is defined by

$$m_{k,i}(\rho) \triangleq \begin{cases} \frac{M_{k,i}\rho M_{k,i}^{\dagger}}{tr(M_{k,i}\rho)}, & \text{if } tr(M_{k,i}\rho) \neq 0, \\ \frac{I}{2^n} & \text{otherwise.} \end{cases}$$

Note that for all $\rho \in \mathfrak{D}(\mathcal{H}_Q)$, $m_{k,i}(\rho) \in \mathfrak{D}(\mathcal{H}_Q)$, as it holds that $tr(m_{k,i}(\rho)) = 1$. Indeed, $tr(M_{k,i}\rho M_{k,i}^{\dagger}) = tr(M_{k,i}^2\rho) = tr(M_{k,i}\rho)$, as $M_{k,i}$ is a projection. Hence $m_{k,i}$ is a map in $\mathfrak{D}(\mathcal{H}_Q) \to \mathfrak{D}(\mathcal{H}_Q)$.

We set $[\![\mathcal{B}]\!] \triangleq \{0,1\}$ and $[\![\mathcal{N}]\!] \triangleq \mathbb{N}$. The classical state is modeled as a (well-typed) *store* s of domain $\mathit{dom}(s)$ mapping each variable $\mathtt{x}$ of type $\mathcal{K}$ to a value in $[\![\mathcal{K}]\!]$. With $\mathtt{Store}$, we denote the set of all such stores. Let $s[\mathtt{x}^{\mathcal{K}} := k]$ with $k \in [\![\mathcal{K}]\!]$ be the store obtained from s by updating the value assigned to $\mathtt{x}$ in the map s. Given a store s, let $[\![-]\!]^s : \mathcal{K}\mathtt{Exp} \to [\![\mathcal{K}]\!]$ be the map associating to each expression $\mathtt{e}$ of type $\mathcal{K}$ and such that $\mathcal{B}(\mathtt{e}) \cup \mathcal{N}(\mathtt{e}) \subseteq \mathit{dom}(s)$, a value in $[\![\mathcal{K}]\!]$, defined in the obvious way. For example $[\![\mathtt{x}]\!]^s \triangleq s(\mathtt{x})$, $[\![n]\!]^s \triangleq n$, $[\![\mathtt{tt}]\!]^s \triangleq 1$, $[\![\mathtt{n}_1 - \mathtt{n}_2]\!]^s \triangleq \max(0, [\![\mathtt{n}_1]\!]^s - [\![\mathtt{n}_2]\!]^s)$, etc.

Let $\downarrow$ be a special symbol for termination. A *configuration* μ, for (extended) statement $\mathtt{stm} \in \mathtt{Stmt} \cup \{\downarrow\}$, store $s \in \mathtt{Store}$, and a quantum state $\rho \in \mathcal{H}_Q$, has the form $(\mathtt{stm}, s, \rho)$. Let $\mathtt{Conf}$ be the set of configurations. A configuration $(\mathtt{stm}, s, \rho)$ is well-formed with respect to the sets of variables B, V, and Q if $\mathcal{B}(\mathtt{stm}) \subseteq B$, $\mathcal{N}(\mathtt{stm}) \subseteq V$, $\mathcal{Q}(\mathtt{stm}) \subseteq Q$, $\mathit{dom}(s) = B \cup V$, and $\rho \in \mathfrak{D}(\mathcal{H}_Q)$. Throughout the paper, we only consider configurations that are well-formed with respect to the sets of variables of the program under consideration.

The operational semantics is described in Figure 4 as a PARS $\to_{\mathrm{Q}}$ over objects in $\mathtt{Conf}$, where terminal objects are precisely the configurations of the shape $(\downarrow, s, \rho)$. The (classical or quantum) state of a configuration can only be updated by the three rules (Exp), (Op), and (Meas). Rule (Exp) updates the classical store wrt the value of the evaluated expression. Rule (Op) updates the quantum state to a new quantum state $\Phi_{U_{\overline{\mathtt{q}}}}(\rho) = U_{\overline{\mathtt{q}}}\rho U_{\overline{\mathtt{q}}}^{\dagger}$, where $U_{\overline{\mathtt{q}}}$ is the unitary operator in $\mathcal{M}(\mathcal{H}_Q)$ computed by extending the quantum gate $\mathtt{U}$ to the entire set of qubits Q. Rule (Meas) performs a measurement on qubit $\mathtt{q}_i$. This rule returns a distribution of configurations corresponding to the two possible outcomes, $k = 0$ and $k = 1$, with their respective probabilities $tr(M_{k,i}\rho)$ and, in each case, updates the classical store and the quantum state accordingly. In the particular case where $tr(M_{k_0,i}\rho) = 0$ for some $k_0 \in \{0,1\}$, $\{tr(M_{k,i}\rho) : (\downarrow, s[\mathtt{x} := k], m_{k,i}(\rho))\}_{k \in \{0,1\}} = \{1 : (\downarrow, s[\mathtt{x} := 1 - k_0], m_{1-k_0,i}(\rho))\}$. Rule (Seq) governs the execution of a sequence of statements $\mathtt{stm}_1\mathtt{;}\ \mathtt{stm}_2$, under the covenant that $\downarrow\ \mathtt{;}\ \mathtt{stm} \triangleq \mathtt{stm}$, for each statement $\mathtt{stm}$. The rule accounts for potential probabilistic behavior when $\mathtt{stm}_1$ performs a measurement and it is otherwise standard. All the other rules are standard.

In a configuration $\mu = (\mathtt{stm}, s, \rho)$, the pair $\sigma \triangleq (s, \rho)$ is called a state. Let $\mathtt{St}^{\mathtt{stm}}$ be the set of states $\sigma, \tau, \ldots$ that are well-formed wrt statement $\mathtt{stm}$. For simplicity, we will denote this set by $\mathtt{St}$ when $\mathtt{stm}$ is clear from the context. To ease the presentation, we sometimes write $(\mathtt{stm}, \sigma)$ for the configuration μ.

We will be interested in expectation-based reasoning on quantum programs. In what follows, we also call functions $f : \mathtt{Conf} \to \mathbb{R}^{+\infty}$ *expectations*, for brevity.

Definition 2. *For a statement* $\mathtt{stm}$ *and* $f : \mathtt{St} \to \mathbb{R}^{+\infty}$, *we overload the notions of expected derivation length and weakest pre-expectation by:*

$$\begin{aligned} &\mathsf{edl}_{\mathtt{stm}} : \mathtt{St} \to \mathbb{R}^{+\infty} && \mathsf{qwp}_{\mathtt{stm}} : (\mathtt{St} \to \mathbb{R}^{+\infty}) \to (\mathtt{St} \to \mathbb{R}^{+\infty}) \\ &\mathsf{edl}_{\mathtt{stm}} \triangleq \lambda\sigma.\mathsf{edl}_{\twoheadrightarrow_{\mathrm{Q}}}(\mathtt{stm}, \sigma) && \mathsf{qwp}_{\mathtt{stm}} \triangleq \lambda f.\lambda\sigma.\mathsf{wp}_{\twoheadrightarrow_{\mathrm{Q}}}(f_{st})(\mathtt{stm}, \sigma), \end{aligned}$$

where $f_{st}(\mathtt{stm}, \tau) = f(\tau)$.

Example 2. Consider the program $\mathtt{Cntoss}$ given Figure 3. In the setting of the program $\mathtt{Cntoss}$, $Q = \{\mathtt{q}\}$, $M_{0,1} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ and $M_{1,1} = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$. On an initial state $\sigma = (s, \rho)$, the reduction starts deterministically as in the classical setting, performing the initialization $\mathtt{x} = \mathtt{tt}$ and $\mathtt{i} = 0$. From there, evaluation reaches the loop $\mathtt{while\ x\ do\ stm}$. At each loop iteration, the loop counter $\mathtt{i}$ is incremented, and the Hadamard gate applied to the quantum variable $\mathtt{q}$. The loop guard is obtained through measuring $\mathtt{q}$.

To see how this is reflected in the semantics, let us first look at an iteration of the loop. If $\mathtt{x}$ was set to false, that is $\mathtt{x}$ holds the value 0, by rule (Wh_0) the loop terminates within one step:

$$\{1 : (\mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}0, \mathtt{i}{:=}i], \rho)\} \xrightarrow{1}\!\!\!\twoheadrightarrow_{\mathrm{Q}} \{1 : (\downarrow, [\mathtt{x}{:=}0, \mathtt{i}{:=}i], \rho)\}. \qquad (0)$$

On the other hand, when $\mathtt{x}$ was previously set to true, the loop executes its body. Precisely, we have:

$$\begin{aligned} &\{1 : (\mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}1, \mathtt{i}{:=}i], \rho)\} \\ &\quad \xrightarrow{1}\!\!\!\twoheadrightarrow_{\mathrm{Q}} \{1 : (\mathtt{i} = \mathtt{i}{+}1;\ \mathtt{q} = \mathtt{H};\ \mathtt{x} = \mathtt{meas\ q};\ \mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}1, \mathtt{i}{:=}i], \rho)\} && (1) \\ &\quad \xrightarrow{1}\!\!\!\twoheadrightarrow_{\mathrm{Q}} \{1 : (\mathtt{q} = \mathtt{H};\ \mathtt{x} = \mathtt{meas\ q};\ \mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}1, \mathtt{i}{:=}i+1], \rho)\} && (2) \\ &\quad \xrightarrow{1}\!\!\!\twoheadrightarrow_{\mathrm{Q}} \{1 : (\mathtt{x} = \mathtt{meas\ q};\ \mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}k, \mathtt{i}{:=}i+1], \Phi_H(\rho))\} && (3) \\ &\quad \xrightarrow{1}\!\!\!\twoheadrightarrow_{\mathrm{Q}} \{p_k : (\mathtt{while\ x\ do\ stm}, [\mathtt{x}{:=}k, \mathtt{i}{:=}i+1], \rho_k))\}_{k \in \{0,1\}}, && (4) \end{aligned}$$

where in the last step, the probability p_k equals $\mathrm{tr}(M_{k,1}\Phi_H(\rho))$, while the normalized quantum state ρ_k is given as $m_{k,1}(\Phi_H(\rho))$. The above reduction is obtained by applying the rules of Figure 4: rule (Wh_1) for reduction (1); rules (Exp) and (Seq) for reduction (2); rules (Op) and (Seq) for reduction (3); and finally rules (Meas) and (Seq) for reduction (4).

For an arbitrary initial quantum state $\rho = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \in \mathfrak{D}(\mathcal{H}_Q)$ (where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ and $tr(\rho) = \alpha + \delta = 1$, $\gamma = \overline{\beta}$, etc.), it follows that

$$p_0 = tr(M_{0,1}H\rho H^\dagger) = tr(\left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)\frac{1}{2}\begin{pmatrix} \alpha+\beta+\gamma+\delta & \alpha-\beta+\gamma-\delta \\ \alpha+\beta-\gamma-\delta & \alpha-\beta-\gamma+\delta \end{pmatrix}) = \frac{1+\beta+\gamma}{2},$$

and that, $p_1 = 1 - p_0 = \frac{1-(\beta+\gamma)}{2}$. Using $\rho_k = \frac{M_{k,1} H \rho H^\dagger M_{k,1}^\dagger}{tr(M_{k,1} H \rho H^\dagger)} = \frac{(M_{k,1}H)\rho(M_{k,1}H)^\dagger}{p_k}$,

$$\rho_0 = \frac{\left(\begin{smallmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 0 \end{smallmatrix}\right)\left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)\left(\begin{smallmatrix} 1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 0 \end{smallmatrix}\right)}{p_0} = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right) \quad \rho_1 = \frac{\left(\begin{smallmatrix} 0 & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} \end{smallmatrix}\right)\left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)\left(\begin{smallmatrix} 0 & 1/\sqrt{2} \\ 0 & 1/\sqrt{2} \end{smallmatrix}\right)}{p_1} = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right).$$

Summarizing (1)–(4) we thus get:

$$\begin{aligned}
&\{1 : (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}i], \left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right))\} \\
&\qquad \xrightarrow{4}\!\!\!\!\rightarrow^{4}_{\mathrm{Q}} \{p_0 \colon (\texttt{while x do stm}, [\texttt{x}{:=}0, \texttt{i}{:=}i+1], \rho_0), \\
&\qquad\qquad p_1 \colon (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}i+1], \rho_1)\}.
\end{aligned}$$

Putting everything together, we have

$$\begin{aligned}
(\texttt{Cntoss}, s, \left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)) &\xrightarrow{2}\!\!\!\!\rightarrow^{2}_{\mathrm{Q}} \{1 : (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}0], \rho)\} \\
&\xrightarrow{4}\!\!\!\!\rightarrow^{4}_{\mathrm{Q}} \{p_0 \colon (\texttt{while x do stm}, [\texttt{x}{:=}0, \texttt{i}{:=}1], \rho_0), \\
&\qquad p_1 \colon (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}1], \rho_1)\} \\
&\xrightarrow{p_0+4p_1}\!\!\!\!\rightarrow^{4}_{\mathrm{Q}} \{p_0 \colon \underline{(\downarrow, [\texttt{x}{:=}0, \texttt{i}{:=}1], \rho_0)}, \\
&\qquad \tfrac{p_1}{2} \colon (\texttt{while x do stm}, [\texttt{x}{:=}0, \texttt{i}{:=}2], \rho_0), \\
&\qquad \tfrac{p_1}{2} \colon (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}2], \rho_1)\} \\
&\xrightarrow{\frac{p_1}{2}+4\frac{p_1}{2}}\!\!\!\!\rightarrow^{4}_{\mathrm{Q}} \{p_0 \colon \underline{(\downarrow, [\texttt{x}{:=}0, \texttt{i}{:=}1], \rho_0)}, \\
&\qquad \tfrac{p_1}{2} \colon \underline{(\downarrow, [\texttt{x}{:=}0, \texttt{i}{:=}2], \rho_0)}, \\
&\qquad \tfrac{p_1}{4} \colon (\texttt{while x do stm}, [\texttt{x}{:=}0, \texttt{i}{:=}3], \rho_0), \\
&\qquad \tfrac{p_1}{4} \colon (\texttt{while x do stm}, [\texttt{x}{:=}1, \texttt{i}{:=}3], \rho_1)\} \\
&\xrightarrow{\frac{p_1}{4}+4\frac{p_1}{4}}\!\!\!\!\rightarrow^{4}_{\mathrm{Q}} \quad \ldots
\end{aligned}$$

where terminal configurations are underlined. This reduction converges to the terminal distribution

$$\mathsf{term}_{\texttt{Cntoss}}(s, \rho) = \{p_0 : (\downarrow, [\texttt{x}{:=}0, \texttt{i}{:=}1], \rho_0)\} + \{\tfrac{p_1}{2^i} : (\downarrow, [\texttt{x}{:=}0, \texttt{i}{:=}i+1], \rho_0)\}_{i \geq 1},$$

with an expected derivation length of

$$\mathsf{edl}_{\texttt{Cntoss}}(s, \left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)) = 2 + 4 + (p_0 + 4p_1) + \sum_{i=1}^{\infty} \frac{5p_1}{2^i} = 7 + 8p_1 = 11 - 4(\beta + \gamma).$$

For expectation $f(s, \rho) \triangleq s(\texttt{i})$, measuring the iteration counter `i`, we have

$$\mathsf{qwp}_{\texttt{Cntoss}}\ f\ (s, \left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)) = p_0 \times 1 + \sum_{i=1}^{\infty} \frac{p_1}{2^i}(i+1) = p_0 + p_1 + 2p_1 = 2 - (\beta + \gamma),$$

that is, the mean value held by `i` holds after execution is $2 - (\beta + \gamma)$. The termination probability is

$$\mathsf{qwp}_{\texttt{Cntoss}}\ (\lambda\sigma.1)\ (s, \left(\begin{smallmatrix} \alpha & \beta \\ \gamma & \delta \end{smallmatrix}\right)) = p_0 \times 1 + \sum_{i=1}^{\infty} \frac{p_1}{2^i} \times 1 = p_0 + p_1 = 1,$$

i.e., the program is almost surely terminating.

3 Weakest Pre-expectations and Arithmetical Hierarchy

In this section, we study the hardness of some natural quantitative problems for weakest pre-expectations and expected derivation length.

Computability-Aimed Restrictions. This subsection is devoted to putting some restrictions on programs and on the considered notion of expectation to overcome the issues of computability, mentioned in the introduction.

Algebraic numbers. Towards this end, our solution is to target a subset of complex numbers, where simple operations like equality are decidable. We consider the set $\overline{\mathbb{Q}}$ of *algebraic numbers*, i.e., complex numbers in $\mathbb{C}$ that are roots of a non-zero polynomial in $\mathbb{Q}[X]$. Let $\mathbb{A} \triangleq \overline{\mathbb{Q}} \cap \mathbb{R}$ be the real closed field of real algebraic numbers in $\mathbb{R}$. The following inclusions trivially hold (i) $\mathbb{N} \subseteq \mathbb{Q} \subseteq \mathbb{A} \subseteq \mathbb{R} \subseteq \mathbb{C}$ and (ii) $\overline{\mathbb{Q}} \subseteq \mathbb{C}$. It was proved in [18, Proposition 2.2] that equality over $\overline{\mathbb{Q}}$ and inequality over $\mathbb{A}$ are decidable using Cohn's representation [12]. It is well-known that the product and sum over $\overline{\mathbb{Q}}$ are computable in polynomial time.

We now restrict the program semantics to matrices and density operators over algebraic numbers. Given a totally ordered set of qubits $Q = \{\mathtt{q}_1, \ldots, \mathtt{q}_n\}$, let $\tilde{\mathcal{H}}_Q$ be the Hausdorff pre-Hilbert space $\overline{\mathbb{Q}}^{2^n}$ (i.e., the completeness requirement on Hilbert spaces is withdrawn) of n qubits defined by $\tilde{\mathcal{H}}_Q \triangleq \otimes_{i=1}^{n} \tilde{\mathcal{H}}_{\mathtt{q}_i}$, with $\tilde{\mathcal{H}}_{\mathtt{q}} \triangleq \overline{\mathbb{Q}}^2$ being the vector space of computational basis $\{|0\rangle, |1\rangle\}$ over the field $\overline{\mathbb{Q}}$. Let $\mathcal{M}(\tilde{\mathcal{H}}_Q)$ and $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$ be the set of matrices and density operators on $\tilde{\mathcal{H}}_Q$, respectively.

Clifford+T gates. For the program semantics to be defined on the space $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$, the considered quantum gates are now restricted to gates whose corresponding unitary operators are in $\mathcal{M}(\tilde{\mathcal{H}}_Q)$, i.e., have a matrix representation over the algebraic numbers. To this end, we consider a restriction to the *Clifford+T gates*: `I`, `X`, `Y`, `Z`, `H`, `S`, `CNOT`, and `T`, whose unitary matrices are given below:

$$I \triangleq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ X \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ Y \triangleq \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix},\ Z \triangleq \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},\ H \triangleq \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$S \triangleq \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 \\ 0 & \mathrm{i} \end{pmatrix},\ CNOT \triangleq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},\ T \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{\mathrm{i}\frac{\pi}{4}} \end{pmatrix}.$$

The Clifford+T fragment is the set of unitary transformations generated by sequential (matrix multiplication) and parallel (Kronecker product) compositions of the gates H, S, $CNOT$, and T. This constitutes a reasonable restriction for unitary operators as it is known to be the simplest approximately universal fragment of quantum mechanics [1].

A central observation is that the superoperator associated with a unitary operator of the Clifford+T fragment is an endomorphism over density operators in $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$.

Lemma 1. *The Clifford+T fragment preserves $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$, i.e., there exist Q and $\overline{\mathtt{q}} \in Q$ such that for each unitary operator U of the Clifford+T fragment $\Phi_{U_{\overline{\mathtt{q}}}} \in \mathfrak{D}(\tilde{\mathcal{H}}_Q) \to \mathfrak{D}(\tilde{\mathcal{H}}_Q)$.*

Notice that, while a restriction to Clifford+T is reasonable in terms of quantum mechanics and universality, our result can be extended by adding any quantum gate preserving the above lemma. For example, the phase shift gate, defined by $P_\varphi \triangleq \left(\begin{smallmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{smallmatrix}\right)$, preserves $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$ whenever $\varphi = r\pi$, for any $r \in \mathbb{Q}$.

Let $\mathtt{Stmt}_{\mathtt{CT}}$ be the set of statements restricted to quantum gates computing Clifford+T unitary operators (hence a subset of $\mathtt{Stmt}$), $\mathtt{St}_{\mathtt{CT}}$ be the set of states whose quantum state is in $\mathfrak{D}(\tilde{\mathcal{H}}_Q)$, and $\mathtt{Conf}_{\mathtt{CT}}$ be the set of well-formed configurations in $(\mathtt{Stmt}_{\mathtt{CT}} \cup \{\downarrow\}) \times \mathtt{St}_{\mathtt{CT}}$. Let $\mathtt{St}_{\mathtt{CT}}^{\mathtt{stm}}$ be the set of states in $\mathtt{St}_{\mathtt{CT}}$ that are well-formed wrt statement $\mathtt{stm}$. Once again, by abuse of notation, we will denote this set by $\mathtt{St}_{\mathtt{CT}}$ when $\mathtt{stm}$ is clear from the context.

A consequence of Lemma 1 is that $\mathtt{Conf}_{\mathtt{CT}}$ is closed under reduction, in the following sense. Let $\mathcal{D}^{\mathrm{fin}}_{\mathbb{A}^+}(A) \subseteq \mathcal{D}(A)$ be the set of *finitely supported* subdistributions δ with algebraic probabilities, i.e., $\delta(a) \in \mathbb{A}^+$ for all $a \in A$.

Lemma 2. *The set $\mathcal{D}^{fin}_{\mathbb{A}^+}(\mathtt{Conf}_{\mathtt{CT}})$ is stable under reduction, more precisely, if $\delta \in \mathcal{D}^{fin}_{\mathbb{A}^+}(\mathtt{Conf}_{\mathtt{CT}})$ and $\delta \overset{c}{\twoheadrightarrow}_{\mathrm{Q}} \varepsilon$, then $\varepsilon \in \mathcal{D}^{fin}_{\mathbb{A}^+}(\mathtt{Conf}_{\mathtt{CT}})$ and $c \in \mathbb{A}^+$.*

Computable expectations. We also restrict the expectation codomain to algebraic numbers. Hence the considered expectations will be functions in $\mathtt{St}_{\mathtt{CT}} \to \mathbb{A}^+$. On its own, this restriction is not sufficient for our concerns, as the set $\mathtt{St}_{\mathtt{CT}} \to \mathbb{A}^+$ is not countable. It implies that there exist expectations in $\mathtt{St}_{\mathtt{CT}} \to \mathbb{A}^+$ that are not computable functions. To resolve this issue, we restrict the space of expectations further to computable ones:

$$\mathtt{E}_{\mathtt{CT}} \triangleq \{f \mid f : \mathtt{St}_{\mathtt{CT}} \to \mathbb{A}^+,\ f \text{ computable}\}.$$

An immediate consequence of Lemma 2 is that $\mathsf{term}_{\mathtt{stm}}(\sigma) \in \mathcal{D}(\mathtt{Conf}_{\mathtt{CT}})$ for any $\mathtt{stm} \in \mathtt{Stmt}_{\mathtt{CT}}$ and $\sigma \in \mathtt{St}_{\mathtt{CT}}$. In consequence, $\mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma$ is well-defined for all $f \in \mathtt{St}_{\mathtt{CT}}$. This justifies that in our treatment below, we restrict expectations to the class $\mathtt{E}_{\mathtt{CT}}$. However, keep in mind that despite Lemma 2, the subdistribution $\mathsf{term}_{\mathtt{stm}}(\sigma)$, obtained at the limit, does not fall within $\mathcal{D}^{\mathrm{fin}}_{\mathbb{A}^+}(A)$. It is neither finite nor are probabilities algebraic ($\mathbb{A}^+$ is not complete). In particular, in general $\mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma$ is a real number, rather than an algebraic one.

Quantitative Problems. We now define formally the quantitative problems that we study.

Testing problems. Some natural quantitative problems related to weakest pre-expectations are to determine for a given program $\mathtt{stm}$, a given state σ, a given expectation f, and a given algebraic number a, whether the corresponding weakest pre-expectation $\mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma$ is smaller than or equal to a. In this setting, it makes sense to consider any possible relation in the set $\{<, \leq, =, \geq, >\} \subseteq \mathcal{P}(\mathbb{A} \times \mathbb{A})$ as one could be interested in finding precise values, (strict) upper- or lower-bounds.

Definition 3. *The* testing problem *sets* $\textsc{Test}_{\mathcal{R}} \subseteq \mathtt{Conf}_{\mathtt{CT}} \times \mathtt{E}_{\mathtt{CT}} \times \mathbb{A}^{+}$*, for* $\mathcal{R} \in \{<, \leq, =, \geq, >\}$*, are defined by:*

$$(\mathtt{stm}, \sigma, f, a) \in \textsc{Test}_{\mathcal{R}} :\Longleftrightarrow (\mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma)\ \mathcal{R}\ a.$$

The consideration of both $\textsc{Test}_{\leq}$ and $\textsc{Test}_{>}$ may seem redundant, as $\textsc{Test}_{>}$ can be viewed as the complement of $\textsc{Test}_{\leq}$. However, it makes perfect sense to distinguish both properties, when considering the corresponding universal problems, as we do in a moment.

Finiteness problem. Another problem of interest consists in checking whether the weakest pre-expectations produces some finitary output.

Definition 4. *The* finiteness problem *set* $\textsc{Test}_{\neq\infty} \subseteq \mathtt{Conf}_{\mathtt{CT}} \times \mathtt{E}_{\mathtt{CT}}$ *is defined by:*

$$(\mathtt{stm}, \sigma, f) \in \textsc{Test}_{\neq\infty} :\Longleftrightarrow \mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma < \infty.$$

Termination problems. We also define two termination problems for almost sure termination and positive almost sure termination:

Definition 5. *The sets of (*positive*)* almost-sure terminating *configurations* $\textsc{Ast} \subseteq \mathtt{Conf}_{\mathtt{CT}}$ *(*$\textsc{Past} \subseteq \mathtt{Conf}_{\mathtt{CT}}$*) are defined by:*

$$(\mathtt{stm}, \sigma) \in \textsc{Ast} :\Longleftrightarrow |\mathsf{term}_{\mathtt{stm}}(\sigma)| = 1$$
$$(\mathtt{stm}, \sigma) \in \textsc{Past} :\Longleftrightarrow \mathsf{edl}_{\mathtt{stm}}(\sigma) < \infty.$$

It is well-known that $\textsc{Past} \subsetneq \textsc{Ast}$, cf. [9].

Universal problems. Another kind of natural problems arises if one tries to check some properties for each possible program input (i.e., for each state σ). We can thus define universal properties for each of the sets described previously.

Definition 6. *The sets of universal testing, finiteness and (positive) a.s. termination problems are defined by:*

$$\begin{aligned}
(\mathtt{stm}, f, g) \in \textsc{UTest}_{\mathcal{R}} \subseteq \mathtt{Stmt}_{\mathtt{CT}} \times \mathtt{E}^2_{\mathtt{CT}} &\Longleftrightarrow \forall \sigma \in \mathtt{St}_{\mathtt{CT}},\ (\mathtt{stm}, \sigma, f, g(\sigma)) \in \textsc{Test}_{\mathcal{R}} \\
(\mathtt{stm}, f) \in \textsc{UTest}_{\neq\infty} \subseteq \mathtt{Stmt}_{\mathtt{CT}} \times \mathtt{E}_{\mathtt{CT}} &\Longleftrightarrow \forall \sigma \in \mathtt{St}_{\mathtt{CT}},\ (\mathtt{stm}, \sigma, f) \in \textsc{Test}_{\neq\infty} \\
\mathtt{stm} \in \textsc{UAst} \subseteq \mathtt{Stmt}_{\mathtt{CT}} &\Longleftrightarrow \forall \sigma \in \mathtt{St}_{\mathtt{CT}},\ (\mathtt{stm}, \sigma) \in \textsc{Ast} \\
\mathtt{stm} \in \textsc{UPast} \subseteq \mathtt{Stmt}_{\mathtt{CT}} &\Longleftrightarrow \forall \sigma \in \mathtt{St}_{\mathtt{CT}},\ (\mathtt{stm}, \sigma) \in \textsc{Past}
\end{aligned}$$

Example 3. We have $\mathtt{Cntoss} \in \textsc{UAst}$ and $\mathtt{Cntoss} \in \textsc{UPast}$, for the program `Cntoss` of Figure 3. Indeed, it was shown in Example 2 that `Cntoss` terminates with probability 1 and a finite expected derivation length. This property holds for any input of the domain. In the same example, we have proven $(\mathtt{Cntoss}, f) \in \textsc{Test}_{\neq\infty}$ for $f(s, \rho) = s(\mathtt{i})$. Indeed, we have shown the stronger property $(\mathtt{Cntoss}, f, g) \in \textsc{Test}_{=}$, where $g(s, \left(\begin{smallmatrix}\alpha & \beta \\ \gamma & \delta\end{smallmatrix}\right)) = 2 - (\beta + \gamma)$.

	Standard		Universal	
	Problem	Class	Problem	Class
Testing	$\textsc{Test}_{>}$	Σ_1^0	$\textsc{UTest}_{>}$	Π_2^0 (‡)
	$\textsc{Test}_{\geq}$	Π_2^0 (‡)	$\textsc{UTest}_{\geq}$	Π_2^0 (‡)
	$\textsc{Test}_{=}$	Π_2^0	$\textsc{UTest}_{=}$	Π_2^0 (‡)
	$\textsc{Test}_{\leq}$	Π_1^0 (‡)	$\textsc{UTest}_{\leq}$	Π_1^0 (‡)
	$\textsc{Test}_{<}$	Σ_2^0	$\textsc{UTest}_{<}$	Π_3^0 (‡)
Finiteness	$\textsc{Test}_{\neq\infty}$	Σ_2^0	$\textsc{UTest}_{\neq\infty}$	Π_3^0 (‡)
Termination	$\textsc{Ast}$	Π_2^0	$\textsc{UAst}$	Π_2^0
	$\textsc{Past}$	Σ_2^0	$\textsc{UPast}$	Π_3^0

Table 1. Completeness results for quantitative problems in the arithmetical hierarchy.

Completeness Results in the Arithmetical Hierarchy. In what follows, we place the introduced quantitative problems within the *arithmetical hierarchy* [34]. The arithmetical hierarchy is a means to classify and relate *undecidable* problems wrt. to their inherent difficulty, measured in terms of the number of (unbounded) quantifier alternations needed to state the problem as a formula in first-order arithmetic, based on a decidable (recursive) predicate.

Reminder on the arithmetical hierarchy. Classes of the arithmetical hierarchy are defined inductively as follows:

$$\Pi_0^0 = \Sigma_0^0 \triangleq \textsc{rec}, \qquad \textsc{rec} \text{ being the class of decidable problems (recursive sets)}$$
$$\Pi_{n+1}^0 \triangleq \{\psi \mid \exists \phi \in \Sigma_n^0,\ \forall \overline{x}.(\psi(\overline{x}) \iff \forall \overline{y}.\phi(\overline{x}, \overline{y}))\},$$
$$\Sigma_{n+1}^0 \triangleq \{\psi \mid \exists \phi \in \Pi_n^0,\ \forall \overline{x}.(\psi(\overline{x}) \iff \exists \overline{y}.\phi(\overline{x}, \overline{y}))\}.$$

For each n, Π_n^0 is the complement of Σ_n^0 (i.e., $\Pi_n^0 = \text{co-}\Sigma_n^0$, and vice versa) and it is a well-known result that Σ_1^0 and Π_1^0 correspond to the classes RE of recursively enumerable (i.e., semi-decidable) problems and co-RE of co-recursively enumerable (i.e., co-semi-decidable) problems, respectively. Given the sets $A \subseteq X$ and $B \subseteq Y$, we write $A \leq_{\mathsf{m}} B$ (A is many-one reducible to B) if there exists a computable function $f : X \to Y$ such that $\forall x \in X,\ x \in A \iff f(x) \in B$. Given a class C of the arithmetical hierarchy and a set A, A is C-*hard* if $\forall B \in \text{C},\ B \leq_{\mathsf{m}} A$. A set A is C-*complete* if $A \in \text{C}$ and A is C-hard. It is well-known that if a set A is C-complete then its complement, noted co-A, is (co-C)-complete.

Results. Table 1 associates the quantum decision problems to the corresponding classes in the arithmetical hierarchy for which we have proven them *complete*, that is, we have proven membership and hardness for the corresponding class. Some of the results may seem surprising. For instance, the testing problem $\textsc{Test}_{>}$, i.e., deciding $\mathsf{qwp}_{\mathsf{stm}}\ f\ \sigma > a$ within the Clifford+T fragment, turns out to be recursive enumerable. It is thus classified identical to the (classical) *halting* *problem $\mathcal{H}$.*[§] Remarkable, through the restriction to the Clifford+T fragment, corresponding problems are ranked within the arithmetical hierarchy identical to their non-quantum counterparts (see [37,24]). This observation holds for all problems apart those marked with (‡) which, to the best of our knowledge, have not been studied in a classical/probabilistic setting. Π^0_2- and Π^0_3-completeness of the universal testing problems, given relations $>$ and $<$ respectively, has been conjectured by Kaminski in his PhD thesis [25] for probabilistic programs.

A crucial observation towards these results is that, restricting to the Clifford+T fragment, the weakest pre-expectation of a program $\mathtt{P}$ can be *approximated* through *computable* transformers $\mathsf{qwp}^{\leq n}_{\mathtt{stm}} : \mathsf{E}_{\mathsf{CT}} \to \mathsf{E}_{\mathsf{CT}}$ that limit execution of $\mathtt{stm}$ to at most $n \in \mathbb{N}$ reduction steps. That is,

$$\mathsf{qwp}^{\leq n}_{\mathtt{stm}}\ f\ \sigma \triangleq \mathbb{E}_{\mathsf{term}^{\leq n}_{\mathtt{stm}}(\sigma)}(f),$$

for $\mathsf{term}^{\leq n}_{\mathtt{stm}}(\sigma)$ the distribution of terminal configurations obtained within n reduction steps, when evaluating $(\mathtt{stm}, \sigma)$. With regards to the above mentioned $\textsc{Test}_{>} \in \Sigma^0_1$ for instance, observe that:

$$\begin{aligned}(\mathtt{stm}, f, \sigma, a) \in \textsc{Test}_{>} &\iff \mathsf{qwp}_{\mathtt{stm}}\ f\ \sigma > a \\ &\iff \lim_{i\to\infty} \mathsf{qwp}^{\leq n}_{\mathtt{stm}}\ f\ \sigma > a \\ &\iff \exists n \in \mathbb{N}, \exists \delta \in \mathbb{A}^+ \setminus \{0\},\ \mathsf{qwp}^{\leq n}_{\mathtt{stm}}\ f\ \sigma \geq a + \delta.\end{aligned}$$

Crucially, the predicate $\mathsf{qwp}^{\leq n}_{\mathtt{stm}}\ f\ \sigma \geq a+\delta$ becomes computable. In essence, this is a consequence of Lemma 2: The n-th step normal form distribution $\mathsf{term}^{\leq n}_{\mathtt{stm}}(\sigma)$ is finite and computable, as f is computable so is thus $\mathsf{qwp}^{\leq n}_{\mathtt{stm}}\ f\ \sigma$. From here, the result follows now as equality on $\mathbb{A}$ is decidable. The proof of this, as well as all completeness proofs listed in Table 1 can be found in the Appendix. The following constitutes our first main result.

Theorem 1. *All completeness results in Table 1 hold.*

4 Quantum Expectation Transformers

In what follows, we are interested in deliniating subclasses of testing problems that lead to decidability. To this end, we now define a notion of *quantum expectation transformer* as a means to compute symbolically the weakest pre-expectation of a program. We first introduce some preliminary notations in order to lighten the presentation.

Notations. For any expression e, $[\![\mathsf{e}]\!]$ is a shorthand notation for the function $\lambda(s,\rho).[\![\mathsf{e}]\!]^s \in \mathtt{St} \to \mathbb{R}^{+\infty}$. We will also use $f[\mathtt{x} := \mathsf{e}]$ for the expectation $\lambda(s,\rho).f(s[\mathtt{x} := [\![\mathsf{e}]\!]^s], \rho)$. Similarly, for a given map $\chi : \mathfrak{D}(\mathcal{H}_Q) \to \mathfrak{D}(\mathcal{H}_Q)$,

[§]In our context the halting set $\mathcal{H}$ can be defined as the class of classical programs and states $(\mathtt{P}, \sigma)$ for which $\mathtt{P}$ is halting on σ.

$$\begin{aligned}
\texttt{qet}[\,\texttt{skip}\,]\{f\} &\triangleq f\\
\texttt{qet}[\,\texttt{x} = \texttt{e}\,]\{f\} &\triangleq f[\texttt{x} := \texttt{e}]\\
\texttt{qet}[\,\texttt{stm}_1\,;\ \texttt{stm}_2\,]\{f\} &\triangleq \texttt{qet}[\,\texttt{stm}_1\,]\{\texttt{qet}[\,\texttt{stm}_2\,]\{f\}\}\\
\texttt{qet}[\,\texttt{if b then stm}_1\ \texttt{else stm}_2\,]\{f\} &\triangleq \texttt{qet}[\,\texttt{stm}_1\,]\{f\} +_{[\![\texttt{b}]\!]} \texttt{qet}[\,\texttt{stm}_2\,]\{f\}\\
\texttt{qet}[\,\texttt{while b do stm}\,]\{f\} &\triangleq \mathsf{lfp}\left(\lambda F.\texttt{qet}[\,\texttt{stm}\,]\{F\} +_{[\![\texttt{b}]\!]} f\right)\\
\texttt{qet}[\,\overline{\texttt{q}} \mathrel{*}= \texttt{U}\,]\{f\} &\triangleq f[\Phi_{U_{\overline{\texttt{q}}}}]\\
\texttt{qet}[\,\texttt{x} = \texttt{meas}\ \texttt{q}_i\,]\{f\} &\triangleq f[\texttt{x} := 0;\, m_{0,i}] +_{p_{0,i}} f[\texttt{x} := 1;\, m_{1,i}].
\end{aligned}$$

Fig. 5. Quantum expectation transformer $\texttt{qet}[\,\cdot\,]\{\cdot\}$

$f[\chi] \triangleq \lambda(s,\rho).f(s,\chi(\rho))$. We will also sometimes group such state modifications, for instance, $f[\texttt{x} := \texttt{e};\, \chi]$ stands for $(f[\texttt{x} := \texttt{e}])[\chi]$ and $f[\texttt{x} := \texttt{e}, \texttt{y} := \texttt{e}']$ stands for $(f[\texttt{x} := \texttt{e}])[\texttt{y} := \texttt{e}']$.

For $p \in \texttt{St} \to [0,1]$ and $f, g \in \texttt{St} \to \mathbb{R}^{+\infty}$, $f +_p g$ denotes the function $\lambda\sigma.p(\sigma)\cdot f(\sigma) + (1 - p(\sigma))\cdot g(\sigma) \in \texttt{St} \to \mathbb{R}^{+\infty}$, similar we use $f \cdot g$ to denote $\lambda\sigma.f(\sigma)\cdot g(\sigma) \in \texttt{St} \to \mathbb{R}^{+\infty}$. Thus, for instance, $f[\texttt{x} := \texttt{x}+1] +_{[\![\texttt{x}=1]\!]} f$ behaves like f, except that $\texttt{x}$ is first incremented when applied to states with classical variable $\texttt{x}$ equal to 1. In correspondence to the normalization of quantum state $m_{k,i}$, we define probabilities $p_{k,i} \triangleq \lambda\rho.tr(M_{k,i}\rho M_{k,i}^{\dagger})$. We overload this function from $\mathfrak{D}(\mathcal{H}_Q)$ to $\texttt{St}$ s.t. $p_{k,i}(s,\rho) = p_{k,i}(\rho)$. In this way, $f[\texttt{x} := 0;\, m_{0,i}] +_{p_{0,i}} f[\texttt{x} := 1;\, m_{1,i}]$ computes precisely the expected value of f on the distribution of states obtained by measuring the i-th qubit and assigning the outcome to classical variable $\texttt{x}$.

Finally, we denote by $\leq$ also the pointwise extension of the order from $\mathbb{R}^{+\infty}$ to functions, that is, $f \leq g$ holds iff $\forall\sigma \in \texttt{St},\ f(\sigma) \leq g(\sigma)$.

Definition 7 (Quantum expectation transformer). *The* quantum expectation transformer *consists in a program semantics mapping expectations to expectations in a continuation passing style*

$$\texttt{qet}[\,\cdot\,]\{\cdot\} : \texttt{Stmt} \to (\texttt{St} \to \mathbb{R}^{+\infty}) \to (\texttt{St} \to \mathbb{R}^{+\infty})$$

and is defined inductively on statements in Figure 5.

This transformer corresponds to the notion of *expected value transformer* of [6] on the Kegelspitze $\texttt{S} = (\mathbb{R}^{+\infty}, +_{\texttt{f}})$, with $+_{\texttt{f}}$ being the forgetful addition. In the case of loops, the least fixed point lfp is defined with respect to the pointwise ordering on the function space $\texttt{St} \to \mathbb{R}^{+\infty}$. Equipped with this ordering, this space forms a ω-CPO. As the quantum transformer can be shown to be ω-continuous, the fixed-point is always defined, cf. [44].

Theorem 2 (Adequacy). *The following holds:*

$$\forall \texttt{stm} \in \texttt{Stmt},\ \forall f : \texttt{St} \to \mathbb{R}^{+\infty},\ \texttt{qwp}_{\texttt{stm}}(f) = \texttt{qet}[\,\texttt{stm}\,]\{f\}\,.$$

continuity	$\mathtt{qet}[\,\mathtt{stm}\,]\{\sup_i f_i\} = \sup_i \mathtt{qet}[\,\mathtt{stm}\,]\{f_i\}$
monotonicity	$f \le g \Rightarrow \mathtt{qet}[\,\mathtt{stm}\,]\{f\} \le \mathtt{qet}[\,\mathtt{stm}\,]\{g\}$
upper invariance	$(\llbracket\neg\mathtt{b}\rrbracket\cdot f \le g \wedge \llbracket\mathtt{b}\rrbracket\cdot\mathtt{qet}[\,\mathtt{stm}\,]\{g\} \le g)\Rightarrow\mathtt{qet}[\,\mathtt{while\ b\ do\ stm}\,]\{f\} \le g$

Fig. 6. Universal laws derivable for the quantum expectation transformer.

Apart from continuity, the quantum expectation transformer satisfies several useful laws, see Figure 6. The *(monotonicity)* Law permits us to reason modulo upper-bounds: actual expectations can be always substituted by upper-bounds. It is in fact an immediate consequence from the *(continuity)* Law, which is defined for any ω-chain $(f_i)_i$. The *(upper invariance)* Law constitutes a generalization of the notion of invariant stemming from Hoare calculus. It is used to find closed-form upper-bounds g to expectations f of loops. The pre-conditions state that g should dominate f on states where the loop would immediately exist, and otherwise, should remain invariant under iteration. It is worth mentioning that this proof rule is not only sound, but also complete, in the sense that any upper-bound satisfies the two constraints. The following example illustrates the use of this rule on the running example.

Example 4. Following Example 2, we over-approximate $\mathtt{qet}[\,\mathtt{Cntoss}\,]\{f\}$, for $f(s,\rho) = s(\mathtt{i})$ the post-expectation measuring the classical variable $\mathtt{i}$.

To this end, observe that the function $g : \mathtt{St} \to \mathbb{R}^{+\infty}$ is an upper-invariant (Figure 6) to the while loop `while x do stm`, given a post-expectation $f : \mathtt{St} \to \mathbb{R}^{+\infty}$. Recall that the loop body `stm` comprises ($\mathtt{i} = \mathtt{i{+}1};\ \mathtt{q} \mathrel{*}= \mathtt{H};\ \mathtt{x} = \mathtt{meas\ q}$). To fulfill the conditions of the *(upper invariance)* Law the following inequalities have to be met:

$$\llbracket\neg\mathtt{x}\rrbracket \cdot f \le g \qquad \llbracket\mathtt{x}\rrbracket \cdot \mathtt{qet}[\,\mathtt{i} = \mathtt{i{+}1};\ \mathtt{q} \mathrel{*}= \mathtt{H};\ \mathtt{x} = \mathtt{meas\ q}\,]\{g\} \le g. \tag{5}$$

By unfolding the definition, we see

$$\begin{aligned}
&\mathtt{qet}[\,\mathtt{i} = \mathtt{i{+}1};\ \mathtt{q} \mathrel{*}= \mathtt{H};\ \mathtt{x} = \mathtt{meas\ q}\,]\{g\}\\
&\quad = \mathtt{qet}[\,\mathtt{i} = \mathtt{i{+}1}\,]\{\mathtt{qet}[\,\mathtt{q} \mathrel{*}= \mathtt{H}\,]\{\mathtt{qet}[\,\mathtt{x} = \mathtt{meas\ q}\,]\{g\}\}\}\\
&\quad = \mathtt{qet}[\,\mathtt{i} = \mathtt{i{+}1}\,]\big\{\mathtt{qet}[\,\mathtt{q} \mathrel{*}= \mathtt{H}\,]\big\{g[\mathtt{x}{:=}0;\ m_{0,1}] +_{p_{0,1}} g[\mathtt{x}{:=}1;\ m_{1,1}]\big\}\big\}\\
&\quad = \mathtt{qet}[\,\mathtt{i} = \mathtt{i{+}1}\,]\big\{g[\mathtt{x}{:=}0;\ m_{0,1}\,;\ \Phi_H] +_{p_{0,1}\cdot\Phi_H} g[\mathtt{x}{:=}1;\ m_{1,1}\,;\ \Phi_H]\big\}\\
&\quad = g[\mathtt{x}{:=}0;\ m_{0,1}\,;\ \Phi_H\,;\ \mathtt{i{:=}i{+}1}] +_{p_{0,1}\cdot\Phi_H} g[\mathtt{x}{:=}1;\ m_{1,1}\,;\ \Phi_H\,;\ \mathtt{i{:=}i{+}1}]\\
&\quad = \lambda(s,\rho).\sum_{k\in\{0,1\}} p_{k,1}(\Phi_H(\rho)) \cdot g(s[\mathtt{x} = k, \mathtt{i{:=}i{+}1}], m_{k,1}(\Phi_H(\rho))).
\end{aligned}$$

By using the identities computed already in Example 2, we thus obtain

$$\mathtt{qet}[\,\mathtt{stm}\,]\{g\}\,(s, \left(\begin{smallmatrix}\alpha & \beta\\ \gamma & \delta\end{smallmatrix}\right)) = \sum_{k\in\{0,1\}} p_k \cdot g(s[\mathtt{x} = k, \mathtt{i{:=}i{+}1}], \rho_k), \tag{6}$$

where, as in Example 2, $p_0 = \frac{1+\beta+\gamma}{2}$, $p_1 = \frac{1-(\beta+\gamma)}{2}$, $\rho_0 = \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)$ and $\rho_1 = \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)$

We claim that $g(s, \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right)) \triangleq s(\mathtt{i}) + s(\mathtt{x}) \cdot (2 - (\beta+\gamma))$ is an upper-bound to the pre-expectation of the while loop wrt. to the post expectation f. To this end, we check (5). The first inequality is trivially satisfied. Concerning the second, notice that by definition,

$$g(s[\mathtt{x}=0, \mathtt{i}{:=}\mathtt{i}{+}1], \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)) = s(\mathtt{i}){+}1 \quad \text{and} \quad g(s[\mathtt{x}=1, \mathtt{i}{:=}\mathtt{i}{+}1], \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)) = s(\mathtt{i}){+}3.$$

By (6) we have

$$\begin{aligned}\mathtt{qet}[\,\mathtt{stm}\,]\{g\}\,(s, \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right)) &= \frac{1+\beta+\gamma}{2}(s(\mathtt{i})+1) + \frac{1-(\beta+\gamma)}{2}(s(\mathtt{i})+3)\\ &= (s(\mathtt{i})+2) - (\beta+\gamma) = g(s, \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right)),\end{aligned}$$

from which now the second constraint follows by case analysis on the value of $\mathtt{x}$. Hence $\mathtt{qet}[\,\mathtt{while\ x\ do\ stm}\,]\{f\} \leq g$ and, by monotonicity (Figure 6),

$$\begin{aligned}\mathtt{qet}[\,\mathtt{Cntoss}\,]\{f\}\,(s, \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right)) &\leq \mathtt{qet}[\,\mathtt{x}=\mathtt{tt};\ \mathtt{i}=0\,]\{g\}\,(s, \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right))\\ &= g([\mathtt{x}{:=}1, \mathtt{i}{:=}0], \left(\begin{smallmatrix}\alpha&\beta\\\gamma&\delta\end{smallmatrix}\right)) = 2 - (\beta+\gamma).\end{aligned}$$

Note that, in this case, the computed bound is exact.

One question of interest is to find the $\mathtt{qet}[\,\cdot\,]\{\cdot\}$ of a given statement. We obtain the following completeness results as a corollary of Theorem 1 and Theorem 2 on the Clifford+T fragment.

Corollary 1. *The following completeness results hold:*

- $\{(\mathtt{stm}, f, g) \in \mathtt{Stmt}_{\mathtt{CT}} \times \mathtt{E}^2_{\mathtt{CT}} \mid \forall\sigma,\ \mathtt{qet}[\,\mathtt{stm}\,]\{f\}\,(\sigma) \ = g(\sigma)\}$ *is* Π^0_2*-complete.*
- $\{(\mathtt{stm}, f, g) \in \mathtt{Stmt}_{\mathtt{CT}} \times \mathtt{E}^2_{\mathtt{CT}} \mid \forall\sigma,\ \mathtt{qet}[\,\mathtt{stm}\,]\{f\}\,(\sigma) \ \leq g(\sigma)\}$ *is* Π^0_1*-complete.*

The same kind of result can be straightforwardly obtained for each of the quantitative problems defined in previous section. All the corresponding sets are undecidable: they are at best (co-)semi-decidable as illustrated by Figure 1. This motivates us for restricting the problem a bit further to find a class of functions for which the quantitative problems for $\mathtt{wp}_{\mathtt{stm}}\ f$ can be decided.

5 Decidability of qet Inference over a Real Closed Field

Corollary 1 illustrates that it is not sufficient to relax the problem of finding the quantum expectation transformer of a given statement to upper-bounds, in order to make it decidable. The undecidability of finding the quantum expectation transformer of a given program is due to two other issues: 1) *Issue 1:* The computation of a fixpoint for $\mathtt{qet}[\,\cdot\,]\{\cdot\}$ in the case of while loops, 2) *Issue 2:* The check for inequalities over functions in $\mathtt{E}_{\mathtt{CT}}$, whose first-order theory is not decidable. This section is devoted to overcoming these two issues, by finding an expressive fragment on which the inference of an upper-bound of the quantum expectation transformer becomes decidable.

$$\begin{aligned}
\texttt{qinf}[\,\texttt{skip}\,]\{F\} &\triangleq F \\
\texttt{qinf}[\,\texttt{x} = \texttt{e}\,]\{F\} &\triangleq F[\texttt{x} := \texttt{e}] \\
\texttt{qinf}[\,\texttt{stm}_1\,;\ \texttt{stm}_2\,]\{F\} &\triangleq \texttt{qinf}[\,\texttt{stm}_1\,]\{\texttt{qinf}[\,\texttt{stm}_2\,]\{F\}\} \\
\texttt{qinf}\begin{bmatrix}\texttt{if}^\ell\ \texttt{b} \\ \texttt{then stm}_1 \\ \texttt{else stm}_2\end{bmatrix}\{F\} &\triangleq X_\ell,\ \text{with side-cond.} \begin{cases}\texttt{b} \vdash \texttt{qinf}[\,\texttt{stm}_1\,]\{F\} \leq X_\ell \\ \neg\texttt{b} \vdash \texttt{qinf}[\,\texttt{stm}_2\,]\{F\} \leq X_\ell\end{cases} \\
\texttt{qinf}\left[\texttt{while}^\ell\ \texttt{b do stm}\right]\{F\} &\triangleq X_\ell,\ \text{with side-cond.} \begin{cases}\texttt{b} \vdash \texttt{qinf}[\,\texttt{stm}\,]\{X_\ell\} \leq X_\ell \\ \neg\texttt{b} \vdash F \leq X_\ell\end{cases} \\
\texttt{qinf}[\,\overline{\texttt{q}} \mathrel{*}= \texttt{U}\,]\{F\} &\triangleq F[\phi_{U_{\overline{\texttt{q}}}}] \\
\texttt{qinf}\left[\,\texttt{x} = \texttt{meas}^\ell\ \texttt{q}_i\,\right]\{F\} &\triangleq X_\ell,\ \text{with side-cond.} \begin{cases}p_{0,i} = 0 \vdash F[\texttt{x} := 1;\, m_{1,i}] \leq X_\ell \\ p_{1,i} = 0 \vdash F[\texttt{x} := 0;\, m_{0,i}] \leq X_\ell \\ p_{k,i} \neq 0 \vdash F[\texttt{x} := 0;\, m_{0,i}] \\ \qquad +_{p_{0,i}} F[\texttt{x} := 1;\, m_{1,i}] \leq X_\ell\end{cases}
\end{aligned}$$

Fig. 7. Term representations of $\texttt{qinf}[\,\cdot\,]\{\cdot\}$ and their corresponding side-conditions.

Symbolic Inference. As a first step towards automated inference, we define a symbolic variant of the quantum expectation transformer in Figure 7. In the case of conditionals, loops, and measurements, we will use fresh variables for expectations; side conditions will guarantee that these variables indeed denote (upper-bounds to) the corresponding expectations. This means that the symbolic version yields correct results only when the expectations assigned to these variables satisfy all the side conditions. By solving the generated constraints, viz., by finding an interpretation of ascribed variables that satisfy the imposed side-conditions, we effectively arrive at an inference procedure overcoming Issue 1.

To formalize this approach, we associate a unique label ℓ with each loop, conditional, and measurement, occurring in the considered program. Notationally, we write $\texttt{while}^\ell\ \texttt{b do stm}$ / $\texttt{if}^\ell\ \texttt{b then stm}_1\ \texttt{else stm}_2$ / $\texttt{meas}^\ell\ \texttt{q}$. Such labels permit us to associate a unique expectation variable X_ℓ to each of these constructs. Given a set of such expectation variables EVar, the set of terms ETerm, upon which the symbolic quantum expectation transformer operates, is defined according to the following grammar:

$$\mathsf{ETerm}\ F, G ::= X \mid F[\texttt{x} := \texttt{e}] \mid F[\chi] \mid F +_p G,$$

where X stand for an arbitrary expectation variable in EVar. As stressed above, X will be used to denote certain expectations wrt. loops, conditionals, and measurements. We have already introduced the notations $F[\texttt{x} := \texttt{e}]$ and $F[\chi]$ to represent updates to the classical and quantum state, respectively. Here, χ will always denote a finite composition of superoperators ϕ_U and measurements $m_{k,i}$. By ensuring that normalization of quantum states $m_{k,i}(\rho)$ is never considered in the degenerate case of a zero-probability measurement $p_{k,i}(\rho)$, it will thereby always be possible to write χ as $\lambda\rho.\frac{M\rho M^\dagger}{tr(N\rho N^\dagger)}$, for some $M \in \mathcal{M}(\tilde{\mathcal{H}}_Q)$ in the Clifford+T fragment. Finally, following the same reasoning, in the barycentric sum $F +_p G$ the probability p is a function in the quantum state, and will always be of general form $\lambda\rho.\frac{tr(M\rho M^\dagger)}{tr(N\rho N^\dagger)}$, for some $M, N \in \mathcal{M}(\tilde{\mathcal{H}}_Q)$ in the Clifford+T fragment. Similar to before, we may group updates such as in $F[\mathtt{x} := \mathtt{e};\ \chi]$.

The symbolic variation of the expectation transformer can now be defined as

$$\mathtt{qinf}[\,\cdot\,]\{\cdot\} : \mathtt{Stmt} \to \mathsf{ETerm} \to \mathsf{ETerm},$$

generating also a set of side-conditions of the shape $\Gamma \vdash F \leq G$, with the intended meaning that G binds F on all input states that satisfy the predicate Γ. The full definition of qinfer is given in Figure 7. As already hinted, the side conditions ensure that introduced variables X_ℓ indeed yield an upper-bound on the corresponding expectation, in the case of conditionals by case-analysis, and in the case of loops via an application of the upper-invariant law from Figure 6. In the case of measurements, $m_{k,i}$ and $p_{k,i}$ are defined exactly as before. Here, we single out the two cases where the probability of a measurement, either $p_{0,i}(\rho) = tr(M_{0,i}\rho) = tr(M_{0,i}\rho M_{0,i}^\dagger)$ or $p_{1,i}(\rho) = 1 - p_{0,i}(\rho)$, is zero. This way, we avoid the case analysis underlying the definition of $m_{k,i}$ and may, wlog., assume that it is indeed of the form $\lambda\rho.\frac{M_{k,i}\rho M_{k,i}^\dagger}{tr(M_{k,i}\rho M_{k,i}^\dagger)}$, with non-zero trace $tr(M_{k,i}\rho M_{k,i}^\dagger)$.

Example 5. In correspondence to Example 4, let us consider the application of the inference procedure on the program $\mathtt{Cntoss}$, wrt. to the post-expectation $f(s,\rho) = s(\mathtt{i})$. We label the loop and measurement with m and w, respectively.

Let X denote the post-expectation f. Unfolding the definition, we see

$$\begin{aligned}\mathtt{qinf}[\,\mathtt{Cntoss}\,]\{X\} &= \mathtt{qinf}[\,\mathtt{x} = \mathtt{tt};\ \mathtt{i} = 0;\ \mathtt{while}^{\mathsf{w}}\ \mathtt{x}\ \mathtt{do}\ \mathtt{stm}\,]\{X\} \\ &= X_{\mathsf{w}}[\mathtt{x}{:=}1;\ \mathtt{i}{:=}0],\end{aligned}$$

generating the side-conditions $\mathtt{x} \vdash X_{\mathsf{m}}[\Phi_H; \mathtt{i}{:=}\mathtt{i}{+}1] \leq X_{\mathsf{w}}$ and $\neg\mathtt{x} \vdash X \leq X_{\mathsf{w}}$. The left-hand side of the first constraint is obtained from

$$\begin{aligned}\mathtt{qinf}[\,\mathtt{stm}\,]\{X_{\mathsf{w}}\} &= \mathtt{qinf}[\,\mathtt{i} = \mathtt{i}{+}1\,]\{\mathtt{qinf}[\,\mathtt{q} \mathrel{*}= H\,]\{\mathtt{qinf}[\,\mathtt{meas}^{\mathsf{m}}\ \mathtt{q}\,]\{X_{\mathsf{w}}\}\}\} \\ &= X_{\mathsf{m}}[\Phi_H\,;\ \mathtt{i}{:=}\mathtt{i}{+}1].\end{aligned}$$

Note that this expansion generates further constraints, this time on X_{m} representing the measurement. Specifically, it yields the following constraints:

$$\begin{aligned}&p_{1-k,1} = 0 \vdash X_{\mathsf{w}}[\mathtt{x}{:=}k; m_{k,1}] \leq X_{\mathsf{m}}, \qquad (\text{for } k \in \{0,1\}),\\ &p_{0,1} \neq 0 \neq p_{1,1} \vdash X_{\mathsf{w}}[\mathtt{x}{:=}0; m_{0,1}] +_{p_{0,1}} X_{\mathsf{w}}[\mathtt{x}{:=}1; m_{1,1}] \leq X_{\mathsf{m}}.\end{aligned}$$

Using the analysis from Example 4, we interpret X_{w} and X_{m} as:

$$\begin{aligned}\alpha(X_{\mathsf{w}}) &\triangleq \lambda(s, \left(\begin{smallmatrix}\alpha & \beta\\ \gamma & \delta\end{smallmatrix}\right)).\ s(\mathtt{i}) + s(\mathtt{x})(2 - (\beta + \gamma)),\\ \alpha(X_{\mathsf{m}}) &\triangleq \lambda(s, \left(\begin{smallmatrix}\alpha & \beta\\ \gamma & \delta\end{smallmatrix}\right)).\ s(\mathtt{i}) + 2 - 2\alpha.\end{aligned}$$

Furthermore, we interpret the input variable X as f, i.e., $\alpha(X) \triangleq \lambda(s,\rho).\ s(\mathtt{i})$. Notice how $\alpha(X_{\mathsf{w}})$ just corresponds to the upper-invariant g derived in Example 4. Using the assignment, it is now standard to check that it is a solution to the five constraints. For instance, considering states $\sigma = (\{\mathtt{i}{:=}n, \mathtt{x}{:=}x\}, \left(\begin{smallmatrix}\alpha & \beta \\ \gamma & \delta\end{smallmatrix}\right))$, the ultimate constraint amount to the implication

$$\alpha \neq 0 \neq \delta \Rightarrow n +_{\alpha} (n+2) \leq n + 2 - 2\alpha,$$

which trivially holds. Finally, recall $\mathtt{qinf}[\,\mathtt{Cntoss}\,]\{X\} = X_{\mathsf{w}}[\mathtt{x}{:=}1\mathtt{;}\ \mathtt{i}{:=}0]$. This term is interpreted as $\lambda(s, \left(\begin{smallmatrix}\alpha & \beta \\ \gamma & \delta\end{smallmatrix}\right)).\ 2-(\beta+\gamma)$, yielding the optimal bound computed in Example 4.

Example 6. Re-consider program $\mathtt{RUS}$ depicted in Figure 1. Here, we are interested in an upper-bound on the number of T-gates, counted by the program variable $\mathtt{i}$. As before, we label the loop and measurement with m and w, respectively. Let

$$\mathtt{stm} = \overbrace{\mathtt{q}_2 = |0\rangle\ \mathtt{;}\ \ldots}^{\mathtt{stm}_0}\mathtt{;}\ \mathtt{x} = \mathtt{meas}^{\mathsf{m}}\ \mathtt{q}_2,$$

be the body of the while loop statement (see Figure 1). We proceed with the analysis backwards. By the rules of Figure 7 it holds that $\mathtt{qinf}[\,\mathtt{stm}_0\,]\{F\} = F[\varPhi\mathtt{;}\ \mathtt{i}{:=}\mathtt{i}{+}2]$ for any F, where $\varPhi$ gives the quantum state updates within $\mathtt{stm}_0$. Unfolding definitions, we have $\mathtt{qinf}[\,\mathtt{RUS}\,]\{X\} = X_{\mathsf{w}}[\mathtt{x}{:=}0\mathtt{;}\ \mathtt{i}{:=}1]$ with $\mathtt{x} \vdash X_{\mathsf{m}}[\varPhi; \mathtt{i}{:=}\mathtt{i}{+}2] \leq X_{\mathsf{w}}$ and $\neg\mathtt{x} \vdash X \leq X_{\mathsf{w}}$, since, by the above observation,

$$\mathtt{qinf}[\,\mathtt{stm}\,]\{X_{\mathsf{w}}\} = \mathtt{qinf}[\,\mathtt{stm}_0\,]\{\mathtt{qinf}[\,\mathtt{x} = \mathtt{meas}^{\mathsf{m}}\ \mathtt{q}_2\,]\{X_{\mathsf{w}}\}\} = X_{\mathsf{m}}[\varPhi\mathtt{;}\ \mathtt{i}{:=}\mathtt{i}{+}2],$$

subject to the following additional constraints stemming from measurements:

$$\begin{aligned} &p_{1-k,2} = 0 \vdash X_{\mathsf{w}}[\mathtt{x}{:=}k; m_{k,2}] \leq X_{\mathsf{m}}, \qquad \text{(for } k \in \{0,1\}\text{)},\\ &p_{0,2} \neq 0 \neq p_{1,2} \vdash X_{\mathsf{w}}[\mathtt{x}{:=}0; m_{0,2}] +_{p_{0,2}} X_{\mathsf{w}}[\mathtt{x}{:=}1; m_{1,2}] \leq X_{\mathsf{m}}. \end{aligned}$$

Taking $\alpha(X) \triangleq \lambda(s,\rho).\ s(\mathtt{i})$ and solving the constraints yields a constant upper bound of $8/3$ on the expected number of T-gates used by the program. This is due to the fact that the probability of the internal measurement is always $\frac{3}{4}$. Note that this bound is tight.

The transformer qinfer can be linked to qet of course only when variables X_ℓ are interpreted in a way that the side conditions generated by infer are met. To spell this out formally, let $\alpha : \mathsf{EVar} \to \mathbb{E}_{\mathsf{CT}}$ be an *assignment* of expectations to variables in EVar, and let $[\![F]\!]^\alpha : \mathbb{E}_{\mathsf{CT}}$ denote the interpretation of $F \in \mathsf{ETerm}$ under α defined in the natural way, e.g., $[\![X_\ell]\!]^\alpha = \alpha(X_\ell)$, $[\![F[\chi]]\!]^\alpha = [\![F]\!]^\alpha[\chi]$, etc.

We say that a constraint $\varGamma \vdash F \leq G$ is *valid under* α if $[\![F]\!]^\alpha(\sigma) \leq [\![G]\!]^\alpha(\sigma)$ holds for all states $\sigma \in \mathtt{St}_{\mathsf{CT}}$ with $\varGamma(\sigma)$. An assignment α is a *solution* to a set of constraints $\mathcal{C}$, if it makes every constraint in $\mathcal{C}$ valid. Finally, we say α is a solution to $\mathtt{qinf}[\,\mathtt{stm}\,]\{f\}$ if it is a solution to the set of constraints generated by $\mathtt{qinf}[\,\mathtt{stm}\,]\{f\}$. We have the following correspondence:

Theorem 3. *For any $\alpha \in \mathsf{EVar} \to \mathtt{E}_{\mathtt{CT}}$, if α is solution to $\mathtt{qinf}[\,\mathtt{stm}\,]\{F\} = G$, then it holds that $\mathtt{qet}[\,\mathtt{stm}\,]\{[\![F]\!]^\alpha\} \leq [\![G]\!]^\alpha$.*

It is worth mentioning that the above procedure could have been defined without restriction to the full space $\mathtt{St} \to \mathbb{R}^{+\infty}$ of expectations. In this case, this symbolic approach is also complete, in the sense that if $\mathtt{qet}[\,\mathtt{stm}\,]\{f\} = g$ then $\mathtt{qinf}[\,\mathtt{stm}\,]\{X\} = G$ for some G such that the side-conditions have a solution α, with $\alpha(X) = f$ and $[\![G]\!]^\alpha = g$. As our main focus is on decidability, however, we have made the choice to restrict ourself to the Clifford+T setting.

Restriction to Polynomials over the Real Closed Field $\mathbb{A}$. We now turn our eyes towards constraint solving, addressing the remaining Issue 2 through restricting the domain of expectations to *polynomials over algebraic numbers.* To be more precise, we consider the following problem.

Definition 8. *Let $E \subseteq \mathtt{E}_{\mathtt{CT}}$ be a class of expectations. The* inference problem *$\textsc{Qinfer}(E) \subseteq \mathtt{Stmt}_{\mathtt{CT}} \times E \times (\mathsf{EVar} \to E)$ is given by*

$$(\mathtt{stm}, f, \alpha) \in \textsc{Qinfer}(E) \iff \alpha[X := f] \text{ is solution to } \mathtt{qinf}[\,\mathtt{stm}\,]\{X\}$$

In the above definition, $(\mathtt{stm}, f, \alpha) \in \textsc{Qinfer}(E)$ is satisfied if the statement $\mathtt{stm}$ has solution $\alpha[X := f]$ wrt. the expectation f. Hence it can be seen as checking whether f is a post-expectation for $\mathtt{stm}$. In particular, any solution $\alpha[X := f]$ constitutes an upper bound on the weakest pre-expectation of f (see Theorem 3). We will now see that $\textsc{Qinfer}(E)$ is decidable, for E the set of *(real algebraic) polynomial expectations* of (arbitrary but fixed) degree d. For states $\mathtt{St}_{\mathtt{CT}}$ over n classical variables $\mathtt{y}_1, \ldots, \mathtt{y}_n$ and m qubits, let $\mathbb{A}^d[\mathtt{St}_{\mathtt{CT}}]$ denotes the class of functions of *polynomial expectations* of the form

$$\lambda(\{\mathtt{y}_i := Y_i\}_{1\leq i\leq n}, (A_{j,k} + \mathrm{i}B_{j,k})_{1\leq j,k\leq 2^m}).\ P, \tag{7}$$

where variables Y_i refer to the classical, and variables $A_{j,k}$ and $B_{j,k}$ refer to the real part and imaginary part, respectively, of each algebraic coefficient in the quantum state. Further, $P \in \mathbb{A}[Y_1, \ldots, Y_n, A_{1,1}, \ldots, A_{2^m,2^m}, B_{1,1}, \ldots, B_{2^m,2^m}]$ is a multivariate polynomial with coefficients in $\mathbb{A}$. The index d refers to the (total) degree of the underlying polynomial P. For instance,

$$\lambda(\{\mathtt{x} := X;\ \mathtt{i} := I\}, \begin{pmatrix} A_{1,1}+\mathrm{i}B_{1,1} & A_{1,2}+\mathrm{i}B_{1,2} \\ A_{2,1}+\mathrm{i}B_{2,1} & A_{2,2}+\mathrm{i}B_{2,2} \end{pmatrix}).\ I + X(2 - (A_{1,2} + A_{2,1})) \in \mathbb{A}^2[\mathtt{St}_{\mathtt{CT}}]$$

One important remark here is that we allow for possibly negative polynomials whereas expectations only output positive real algebraic numbers. Consequently, some side conditions are put on the admissible coefficients $A_{j,k}$ and $B_{j,k}$ of the input density matrix to preserve this condition (the matrix is positive, has trace 1, is hermitian). For example, $\sum_{i=1}^{2^m} A_{i,i} = 1$, $\sum_{i=1}^{2^m} B_{i,i} = 0$ (trace is 1) and $\forall i,\ k,\ A_{i,k} = A_{k,i}$ and $B_{i,k} = -B_{k,i}$ (self-adjointness). One can easily check that the expectations defined in Example 5 are in $\mathbb{A}^d[\mathtt{St}_{\mathtt{CT}}]$, for $d \geq 1$.

The restriction to polynomials is made on purpose, as quantifier elimination is decidable in the theory of real closed fields, a well known result due to Tarski and Seidenberg. Recall that the theory of real closed fields is the first-order theory in which the primitive operations are multiplication, addition, the order relation $\leq$, and the constants 0 and 1. Consequently, the only numbers that can be defined are the real algebraic numbers. Specifically, we will make use of the following result, quantifying the complexity of the quantifier elimination decision procedure as a function exponential in number of variables, and double-exponential in the number of quantifier alternations.

Proposition 1 ([21, Theorem 6]). *Let $\mathbf{A}$ be an integral ring over a real closed field $\mathbf{R}$. Let $\psi = Q_1\vec{x}_1.Q_2\vec{x}_2.\cdots Q_l\vec{x}_l.\ \phi$ be a formula in prenex-normal form, where $\forall k,\ Q_k \in \{\forall, \exists\}$, $Q_k \neq Q_{k+1}$, and ϕ is a quantifier-free formula over i variables and j atomic propositions of the shape $P \geq 0$, each P being a polynomial of degree at most d with coefficients in $\mathbf{A}$. There exists an algorithm computing a quantifier-free formula equivalent to ψ in time $O(|\psi|) \cdot (jd)^{i^{O(l)}}$.*

As $\mathbb{A}$ constitutes both an integral ring and a real closed field, the above theorem is in particular applicable taking $\mathbf{A} = \mathbf{R} \triangleq \mathbb{A}$. In the particular case where ψ is a closed formula, the resulting quantifier-free formula is simply a Boolean combination of inequalities over constants from $\mathbb{A}$. Since we already observed that these can be decided in polynomial time, the above proposition thus implies that validity of ψ is decidable under the given time bound.

By restricting assignment α to polynomial expectations, it becomes decidable to check that α is a solution to a given constraint set C. Indeed, under such a polynomial assignment α, a constraint $\Gamma \vdash F \leq G$ becomes expressible as a formula in the theory of real closed field $\mathbb{A}$. By letting α range over polynomial expectations with undetermined coefficients, we can this way arrive at the main decidability result of this section.

Theorem 4. *For any degree $d \in \mathbb{N}$, $d \geq 1$, the problem* $\textsc{Qinfer}(\mathbb{A}^d[\mathtt{St}_{\mathtt{CT}}])$ *is decidable in time $2^{2^{dO(n)}}$, where n is the size of the considered program.*

Practical Algorithm. Theorem 4 established a computable algorithm on the inference of upper bounds on weakest pre-expectation on quantitative program properties of any given mixed classical-quantum program. Nevertheless, the complexity of this algorithm — double-exponential in the program size — is forbiddingly high. In order to turn this procedure into a practical algorithm, we have to tame this inherent complexity. For this, significant further restrictions on the class of bounding functions are necessary. We propose to proceed as follows. (1) *Bounding functions:* in (7) we restricted the class of expectations to polynomials, which in turn yield a bound on the weakest pre-expectation. Based on an analysis of concrete examples considered in the literature (e.g., [30,6]), this can be tightened further to degree 2 polynomials. (2) *Approximate solutions*: Theorem 4 rests upon (the decidability) of quantifier elimination. Thus the constraints C induced through the symbolic inference of $\mathtt{qinf}[\,\mathtt{stm}\,]\{X\} = G$

($G, X \in \mathsf{ETerm}$) are solved exactly. Over-approximation, however, suffices, if we are only interested in soundness of the inference mechanism.

The restriction of the class of bounding functions is in essence a question of applicability of the automation, taking into account particular use-cases. With respect to approximate solutions, we observe that the actual constraints C considered have at most one quantifier alternation and admit a quantifier prenex of the form $\exists^*\forall^*$, that is, a sequence of existential quantifier follows by a sequence of universal quantifiers. Roughly speaking the existential quantifiers refer to the inference of coefficients in the bounding polynomials, while the universal quantifiers refer to program variables. It is well-known that universal quantification in optimization problems can be turned into existential quantification, like Farka's lemma or generalizations thereof, cf. [38,19]. (E.g., [7,29] for instances of this approach for the inference of expected program costs.)

Summarizing, the inference mechanism detailed in Section 5 can be over-approximated to generate purely existential constraints. The latter can be effectively solved via SMT. We expect that (full) automation of the inference mechanism can capitalize on these ideas. Working out the details and in particular implementation of an effective prototype is subject to future work.

6 Conclusion and Future Work

We have studied the complexity and inference of techniques for obtaining qualitative program properties. One particular property of interest would be the cost of quantum programs, that is average time, average number of gates, mean value of a variable, etc. We show that these problems were undecidable in general by placing them in the arithmetic hierarchy and saw that inference could become decidable on a restricted fragment: quantum gates in Clifford+T and a function space with a decidable theory (polynomials of bounded degree over a real closed field). Further, we sketch how the latter can be transformed into an efficient synthesis method.

Many open questions remain. The studied notion of expectation transformer describes *local* properties of the quantum state, while it would be interesting to extend this technique to the *global* state so as to study a mixed state in a quantum-only setting (without classical variables and stores). Another question of interest is to what extent a characterization of the quantum class ZBQP, the class of problems computed by quantum programs in polynomial expected runtime, could be obtained using this tool.

Acknowledgments. This work is supported by the HORIZON 2020 project NEASQC and by the Inria associate team TC(Pro)^3. It is also supported by the Plan France 2030 through the PEPR integrated project EPiQ ANR-22-PETQ-0007 and the HQI initiative ANR-22-PNCQ-0002, the ANR PRC project PPS ANR-19-CE48-0014, as well as FWF Project AUTOSARD P 36623.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Physical Review A **70**(5), 052328 (2004)
2. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: 20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings. pp. 249–258. IEEE Computer Society (2005). https://doi.org/10.1109/LICS.2005.1
3. Arute, F., Arya, K., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**, 505–510 (2019). https://doi.org/10.1038/s41586-019-1666-5
4. Avanzini, M., Barthe, G., Lago, U.D.: On continuation-passing transformations and expected cost analysis. Proc. of the ACM on Programming Languages **5**(ICFP), 1–30 (2021). https://doi.org/10.1145/3473592
5. Avanzini, M., Lago, U.D., Yamada, A.: On probabilistic term rewriting. Science of Computer Programming **185** (2020). https://doi.org/10.1016/j.scico.2019.102338
6. Avanzini, M., Moser, G., Péchoux, R., Perdrix, S., Zamdzhiev, V.: Quantum expectation transformers for cost analysis. In: LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022. pp. 10:1–10:13 (2022). https://doi.org/10.1145/3531130.3533332
7. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. Proc. of the ACM on Programming Languages **4**(OOPSLA), 172:1–172:30 (2020). https://doi.org/10.1145/3428240
8. Avanzini, M., Moser, G., Schaper, M.: Automated expected value analysis of recursive programs. Proceedings of the ACM on Programming Languages **7**(PLDI), 1050–1072 (2023). https://doi.org/10.1145/3591263
9. Bournez, O., Garnier, F.: Proving Positive Almost-Sure Termination. In: Proc. of RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer (2005). https://doi.org/10.1142/S0129054112400588
10. Bravyi, S., Kitaev, A.: Universal quantum computation with ideal clifford gates and noisy ancillas. Physical Review A **71**(2), 022316 (2005). https://doi.org/10.1103/PhysRevA.71.022316
11. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L., Yen, D.D.: An automata-based framework for verification and bug hunting in quantum circuits. Proceedings of the ACM on Programming Languages **7**(PLDI), 1218–1243 (2023). https://doi.org/10.1145/3591270
12. Cohn, P.M.: Further algebra and applications. Springer Science & Business Media (2002)
13. Dijkstra, E.W.: A discipline of programming. Prentice-Hall Englewood Cliffs (1976)
14. Endrullis, J., Geuvers, H., Zantema, H.: Degrees of undecidability in term rewriting. In: Grädel, E., Kahle, R. (eds.) Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5771, pp. 255–270. Springer (2009). https://doi.org/10.1007/978-3-642-04027-6_20
15. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: Proto-quipper with dynamic lifting. POPL **7**, 309–334 (2023)
16. Gosset, D., Kliuchnikov, V., Mosca, M., Russo, V.: An algorithm for the t-count. Quantum Information & Computation **14**(15-16), 1261–1276 (2014). https://doi.org/10.26421/QIC14.15-16-1

17. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Performance Evaluation **73**, 110–132 (2014). https://doi.org/10.1016/j.peva.2013.11.004
18. Halava, V., Harju, T., Hirvensalo, M., Karhumäki, J.: Skolem's problem - on the border between decidability and undecidability. Tech. Rep. 683, Turku Center for Computer Science (2005)
19. Handelman, D.: Representing Polynomials by Positive Linear Functions on Compact Convex Polyhedra. PJM **132** (1), 35–62 (1988). https://doi.org/10.2140/pjm.1988.132.35
20. Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. Physical Review Letters **103**, 150502 (2009). https://doi.org/10.1103/PhysRevLett.103.150502
21. Heintz, J., Roy, M.F., Solernó, P.: Sur la complexité du principe de tarski-seidenberg. Bulletin de la Société mathématique de France **118**(1), 101–126 (1990)
22. Hillmich, S., Zulehner, A., Kueng, Richardand Markov, I.L., Wille, R.: Approximating decision diagrams for quantum circuit simulation. ACM Trans. Quantum Comput. **3**(4), 1–21 (2022)
23. Jia, X., Kornell, A., Lindenhovius, B., Mislove, M.W., Zamdzhiev, V.: Semantics for variational quantum programming. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498687
24. Kaminski, B.L., Katoen, J.P.: On the hardness of almost-sure termination. In: MFCS 2015, Part I. pp. 307–318. LNCS, Springer (2015). https://doi.org/10.1007/978-3-662-48057-1_24
25. Kaminski, B.L.: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019), http://publications.rwth-aachen.de/record/755408
26. Kaminski, B.L., Katoen, J.P.: A weakest pre-expectation semantics for mixed-sign expectations. In: LICS 2017. pp. 1–12. IEEE (2017). https://doi.org/10.1109/LICS.2017.8005153
27. Kaminski, B.L., Katoen, J., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run-times of probabilistic programs. In: ESOP 2016. LNCS, vol. 9632, pp. 364–389. Springer (2016). https://doi.org/10.1007/978-3-662-49498-1_15
28. Kozen, D.: A probabilistic PDL. Journal of Computer and System Sciences **30**(2), 162–178 (1985)
29. Leutgeb, L., Moser, G., Zuleger, F.: Automated Expected Amortised Cost Analysis of Probabilistic Data Structures. In: Proc. of 34th CAV. LNCS, vol. 13372, pp. 70–91 (2022). https://doi.org/10.1007/978-3-031-13188-2_4
30. Liu, J., Zhou, L., Barthe, G., Ying, M.: Quantum weakest preconditions for reasoning about expected runtimes of quantum programs (2022)
31. Malherbe, O., Díaz-Caro, A.: Quantum control in the unitary sphere: Lambda-s1 and its categorical model. Logical Methods in Computer Science **18**(3) (2022)
32. McIver, A., Morgan, C.: Abstraction, refinement and proof for probabilistic systems. Springer Science & Business Media (2005)
33. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. ACM SIGPLAN Notices **53**(4), 496–512 (2018). https://doi.org/10.1145/3296979.3192394
34. Odifreddi, P.: Classical recursion theory: The theory of functions and sets of natural numbers. Elsevier (1992)
35. Olmedo, F., Díaz-Caro, A.: Runtime analysis of quantum programs: A formal approach. In: PLanQC 2020 (2020)

36. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: SAS. pp. 270–282. LNCS (2008). https://doi.org/10.1007/978-3-540-69166-2_18
37. Schnabl, A., Simonsen, J.G.: The exact hardness of deciding derivational and runtime complexity. In: Proc. 20th CSL. LIPIcs, vol. 12, pp. 481–495 (2011). https://doi.org/10.4230/LIPIcs.CSL.2011.481
38. Schrijver, A.: Theory of linear and integer programming. Wiley (1999)
39. Selinger, P.: Towards a quantum programming language. Mathematical Structures in Computer Science **14**(4), 527–586 (2004). https://doi.org/10.1017/S0960129504004256
40. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM Journal on Computing **13**(2), 292–314 (1984)
41. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Review **41**(2), 303–332 (1999). https://doi.org/10.1137/S0036144598347011
42. Weihrauch, K.: Computable analysis: an introduction. Springer Science & Business Media (2012)
43. Wille, R., Van Meter, R., Naveh, Y.: IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1234–1240 (2019). https://doi.org/10.23919/DATE.2019.8715261
44. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series, MIT Press (1993)

Reconciling Partial and Local Invertibility

Anders Ågren Thuné[1,2], Kazutaka Matsuda[2](✉), and Meng Wang[3]

[1] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden.
athune@kth.se
[2] Tohoku University, Aramaki Aza-aoba 6-3-09, Aoba-ku, Sendai 980-8579, Japan.
kztk@tohoku.ac.jp
[3] University of Bristol, Bristol BS8 1TH, United Kingdom.
meng.wang@bristol.ac.uk

Abstract. Invertible programming languages specify transformations to be run in two directions, such as compression/decompression or encryption/decryption. Two key concepts in invertible programming languages are *partial invertibility* and *local invertibility*. Partial invertibility lets invertible code be parameterized by the results of non-invertible code, whereas local invertibility requires all code to be invertible. The former allows for more flexible programming, while the latter has connections to domains such as low-energy computing and quantum computing. We find that existing approaches lack a satisfying treatment of partial invertibility, leaving the connection to local invertibility unclear.
In this paper, we identify four core constructs for partially invertible programming, and show how to give them a locally invertible interpretation. We show the expressiveness of the constructs by designing the functional invertible language Kalpis, and show how to give them a locally invertible semantics using the novel arrow combinator language rrArr—the key idea is viewing partial invertibility as an invertible effect. By formalizing the two systems and giving Kalpis semantics by translation to rrArr, we reconcile partial and local invertibility, solving an open problem in the field. All formal developments are mechanized in Agda.

Keywords: Reversible computation · Arrows · Partial invertibility · Domain-specific languages.

1 Introduction

An *invertible* computation can be run in two ways: forward in the conventional way, or backward to recover an input given the output. Such processes appear frequently and prominently in a variety of contexts, enabling the shape of information to be adapted to different purposes, while preserving the essential content. For instance, (lossless) compression shrinks the size of a piece of information to facilitate efficient storage, encryption transforms it to be inaccessible to third parties, and serialization reshapes it to enable storage or transmission. The property of invertibility is crucial, as it guarantees that the data can always be refit to its original purpose.

For example, consider the function *autokey* below, which computes a variant of the Autokey cipher (see *e.g.,* [50]). The cipher takes a primer character k, and interprets it as an integer (*e.g.,* `'A'` $\mapsto 0$, `'B'` $\mapsto 1, \ldots,$ `'Z'` $\mapsto 25$) determining a shift to apply to the first element of the input. Each consecutive character in the input is similarly shifted by the amount given by its predecessor. For instance, *autokey* `'F'` `"HELLO"` = `"CXHAD"`, as `'F'` represents a (cyclic left) shift of 5 characters, mapping `'H'` to `'C'`, and `'H'` a shift of 7 characters, mapping `'E'` to `'X'`, and so on.

```
autokey :: Char → [Char] → [Char]
autokey k []      = []
autokey k (h : t) =
  shift (chrToInt k) h : autokey h t
```

```
autokey′ :: Char → [Char] → [Char]
autokey′ k []        = []
autokey′ k (h′ : t′) =
  let h = shift (−(chrToInt k)) h′
  in  h : autokey′ h t′
```

The corresponding decryption function *autokey′* is given to the right, and shifts backward to restore the original input. We assume *shift* : Int → Char → Char performing the cyclic shift is previously defined. This is a simple example, but it serves as a toy model of more advanced encryption schemes and has a few interesting features which we highlight momentarily.

In traditional unidirectional languages, each direction of an invertible algorithm has to be specified separately in this way, and there is no easy way of ensuring that the two programs really constitute each other's inverses. Furthermore, there is a maintenance concern—when one direction is updated, the other has to be updated accordingly. An alternative, more scalable approach is to let a single program denote both directions at the same time—intuitively, the inverse is derived by "reading the original code right-to-left". *Invertible programming languages* implement this approach, letting each program be executed in either of two directions, which are guaranteed to form a pair of inverse functions. Some examples of invertible languages include Janus [35,53], R [17], Inv [43], *Π* [10,26], RFun [54], Theseus [27], CoreFun [25] and SPARCL [39,40].

These languages traditionally require each individual step of computation to be invertible, which can be ensured, *e.g.,* by providing a set of invertible combinators as basic building blocks, or by imposing various syntactic restrictions. This form of *local invertibility* has several benefits, in addition to being a simple foundation for building programming languages. For example, it was observed early on that discarding information fundamentally results in heat dissipation, meaning that a machine executing only invertible instructions could in principle operate at lower energy levels than a conventional computer [32]. Moreover, locally invertible languages serve as a foundation when considering other domains with similar requirements, such as quantum computing, where computations are composed of individually invertible *quantum gates* along with irreversible *measurements* [22,48]. Despite these benefits, the local flavor of invertibility severely limits the flexibility of the programmer. In particular, our example function *autokey* is not actually invertible up front! The case *autokey* k [ ] = [ ] discards the value of k, which means we cannot simply read the definition right-to-left. Of course, the primer k is not intended to be treated as part of the invertible input to *autokey*, but rather as a parameter determining the bijection between input and output strings. However, this cannot be naturally expressed in a language adhering strictly to the (locally) invertible paradigm, where the parameter would need to be preserved in the result.

The property of becoming invertible when some parameters are fixed is known as *partial invertibility* [39, 40, 44, 47], and many previous languages offer some form of support for partially invertible definitions. However, the level of support varies from more limited (*e.g.,* [25, 27, 35]) to more complete (*e.g.,* [39, 40]), and the previous work largely lacks a systematic treatment. The case of *autokey* is especially tricky, since its invertible input h flows to the unidirectional parameter k in the recursive call. To our knowledge, only Sparcl [39, 40] handles cases like this in a systematic way, but it does so through an advanced language foundation quite different from that of traditional invertible languages, and its connection to the locally invertible paradigm is not well-understood. Thus, it is an open question whether it is possible to support fully expressive partial invertibility while maintaining a compositional locally invertible interpretation.

It is theoretically known that any (partially) invertible computation can be simulated in a locally invertible system [8]; however, this simulation gives poor control over the invertible behavior and is inefficient in both time and space. There has been research on inversion of arbitrary programs (*e.g.,* [41, 44, 49]), and on logic languages with no fixed direction of execution, like Prolog and Curry, which use (lazy) generate-and-test to find inputs corresponding to a given output [4]. Yet, these approaches lack the guarantee of invertibility, which is the main motivation of an invertible language.

1.1 Contributions and Organization

In this paper, we identify a core set of constructs for partially invertible programming, and explain them in terms of a locally invertible semantics. These constructs are sufficient to allow expressive partially-invertible and higher-order computation, thus solving an open problem in the invertible programming literature. The constructs include (1) partially invertible branching, (2) *pinning* invertible inputs, (3) partially invertible composition, and (4) abstraction and application of invertible computations.

We demonstrate the above findings by designing and formalizing two systems based on these constructs, Kalpis[4] and rrArr. Kalpis is a typed functional programming language accommodating expressive partially-invertible and higher-order computation, and rrArr is an arrow combinator language intended to capture the essence of partially invertible programs. Kalpis is given semantics via rrArr, which captures partial invertibility as an effect on top of 'pure' invertible computations, intuitively adjoining a parameter to an invertible function, analogously to the reader monad in unidirectional computation. By interpreting terms of Kalpis as parameterized bijections, we are able to give a

[4] The name stands for "Kalpis—an Arrow-based Locally and Partially Invertible System".

translation into rrArr combinators, giving a compositional embedding into a locally invertible setting. Thus, we present a simple and rigorous take on partial invertibility which bridges the gap between previous work in the field.

The core constructs for partial invertibility that we present are not new per se, and the features of Kalpis largely coincide with those of Sparcl [39, 40]. However, the goal of this paper is not to present Kalpis as such, but rather to describe partial invertibility from first principles and give a simpler semantics which is compatible with local invertibility. There are key technical differences between the two languages, and the fact that they are still similar should be taken as a sign that we have achieved our goal without a significant loss of expressiveness.

In summary, our main contributions are:

- We identify a core set of partially invertible programming constructs (Section 2), which we demonstrate to be sufficient to achieve a level of expressiveness similar to the state-of-the-art.
- We showcase the constructs through the design of the invertible functional language Kalpis, including a formal type system and operational semantics (Section 3).
- We present rrArr, an extension of the irreversibility effect [26] and the reversible reader [23] (Section 4) as a core calculus for partially invertible computation with a locally invertible interpretation.
- We give a compositional translation from Kalpis into rrArr (Section 5).
- We prove type safety and invertibility properties (Section 3), and prove the correctness of the arrow translation (Sections 4 and 5).
- Our developments come with a formalization in Agda including proofs of all theorems,[5] and a prototype implementation of Kalpis.[6]

Section 6 discusses the results in relation to previous work, and Section 7 concludes.

2 Constructs for Partially Invertible Programming

In this section, we introduce a set of core constructs for partially invertible programming and explain their intuitive idea using programming examples in our partially invertible language Kalpis, which we introduce formally in Section 3. The constructs include (1) partially invertible branching, (2) *pinning* invertible inputs, (3) partially invertible composition, and (4) abstraction and application of invertible computations. We explain them each in turn, and show how they can be understood as operations on *parameterized bijections*, which we exploit in later sections to embed them into a locally invertible setting.

These constructs act as a form of glue, allowing invertible and unidirectional computations to be run in tandem. Thus, we also assume some traditional invert-

[5] https://git.sr.ht/~aathn/kalpis-agda
[6] https://git.sr.ht/~aathn/kalpis

ible constructs taken from the existing literature, like invertible pattern matching, which we briefly explain where necessary.

2.1 Partially Invertible Branching

As a first example, we define partially invertible addition. In particular, the function $x \mapsto x + n$ has inverse $x \mapsto x - n$ for any $n \in \mathbb{N}$. KALPIS supports recursive type definitions, and we can define the naturals as follows.

```
data Nat = Z | S Nat
```

Now, addition is implemented naturally by the following function *add*, taking an n to produce the corresponding bijection.

```
sig   add : Nat → Nat ↔ Nat
def•  add n x =
   case n of
      Z   → x
      S n → S (add n ◇ x)
```

The language uses a functional syntax, and features elements typical to invertible programming: a bijection type $A \leftrightarrow B$, bijection definition **def$^\bullet$**, and bijection application $f \diamond x$. The functional types associate to the right, so the type of

$$add : \mathsf{Nat} \to \mathsf{Nat} \leftrightarrow \mathsf{Nat}$$

indicates a partially invertible function taking a Nat to produce a bijection Nat $\leftrightarrow$ Nat. The **case** form showcases our first core construct, *partially invertible branching*. If n is zero, x is returned unchanged, and otherwise S is applied to the result of a recursive computation. The resulting function appends n copies of S to x in the forward direction, or peels them off in the backward direction.

What is interesting is that **case** results in a loss of information: without prior knowledge of n, it is impossible to determine which branch to choose when executing backwards. This corresponds to the fact that one cannot uniquely determine n and x given $y = n + x$. However, when n is fixed beforehand, we can refer to its value regardless of executing forwards or backwards, which is what motivates the **case** construct. For example, we get the following results when applying *add* to some example inputs, where the primitive operator $(\cdot)^\dagger : (A \leftrightarrow B) \to (B \leftrightarrow A)$ lets us compute the inverse.

```
-- 1 + 2 = 3                     -- 3 − 2 = 1
> add (S (S Z)) ◇ S Z            > (add (S (S Z)))† ◇ S (S (S Z))
S (S (S Z))                      S Z
```

As the type Nat $\leftrightarrow$ Nat requires, the argument x in the definition of *add* must be treated *linearly*, *i.e.,* must be used *exactly once* in any successful evaluation (see *e.g.,* [51]) in order to ensure invertibility. For instance, changing the first case above to $\mathsf{Z} \to \mathsf{Z}$ gives an error, as x is unused in the case body. Indeed, if x is never used, there is no way to recover its value in the backward direction. While allowing *more* than one use does not directly prevent invertibility, it requires implicit copying of values, which may induce unintended runtime failures in the backward execution. Similarly, we cannot branch on x using **case** for the reasons mentioned above; instead, an invertible **case**$^\bullet$ form is available, explained later.

Note that *add* is not a total function: *e.g.,* the application $(add\ (\mathsf{S}\ \mathsf{Z}))^\dagger \diamond \mathsf{Z}$ will try to peel an S when there is none, resulting in a runtime error.[7] The guarantee given by KALPIS is that *whenever* evaluating a bijection f on argument v gives v' in the forward direction, then evaluating f on v' gives v in the backward direction, and vice versa (this is made formal in Section 3).

Mathematically, *add* represents a *parameterized bijection*, a family of (partial) one-to-one mappings $f_n : \mathbb{N} \to \mathbb{N}$ (such that $f_n(x) = x + n$). This view will underpin our explanation of partially invertible computations in later sections, and each of the core constructs in this section can also be understood from this viewpoint. Seen from this perspective, the **case** construct allows definitions of the form

$$f_n(x) = \begin{cases} g_n(x) & \text{if } n = 0 \\ h_n(x) & \text{otherwise} \end{cases},$$

where g and h are also parameterized bijections.

2.2 Pinning Invertible Inputs

As a second example, we consider a program *fib* computing pairs of Fibonacci numbers (defined by the equations $F_0 = F_1 = 1$ and $F_{n+1} = F_n + F_{n-1}$ for $n > 0$), a classic in the invertible programming literature (*e.g.,* [18, 53]). We can compute *fib* n by case distinction on n; if $n = 0$, we return (F_0, F_1), and otherwise we recursively obtain $fib\ (n-1) = (F_{n-1}, F_n)$, with which we compute the next pair $(F_n, F_n + F_{n-1})$.

However, if we try to implement this algorithm invertibly using the function *add* above, we encounter an issue: we cannot make the call $add\ F_n \diamond F_{n-1}$, as *add* does not treat its first argument invertibly. Since F_n comes from the invertible input n, we need an operation that is properly invertible in both inputs. To this end, we can define an invertible addition add' such that $add' \diamond (x, y) = (x, x+y)$. By preserving a copy of x in the output, the same x can be used to recover y by subtraction in the inverse direction. Indeed, $add' \diamond (F_n, F_{n-1})$ gives just the result we need. In KALPIS, add' can be derived from *add* automatically using our second core construct, *pin*.

$$\begin{array}{ll} \textbf{sig} & add' : (\mathsf{Nat}, \mathsf{Nat}) \leftrightarrow (\mathsf{Nat}, \mathsf{Nat}) \\ \textbf{def}^\bullet & add'\ (x, y) = pin\ add \diamond (x, y) \end{array}$$

Here, the operator $pin : (c \to a \leftrightarrow b) \to (c, a) \leftrightarrow (c, b)$ lifts a partially invertible function to operate on invertible data; we refer to this as *pinning*

[7] The loss of totality is unavoidable in order to achieve r-Turing completeness [5], *i.e.,* the ability to define all computable bijections.

the invertible input x, allowing it to be used in a unidirectional position. This construct (inherited from Sparcl [39,40]) is crucial in practical programming, as it lets unidirectional computations depend on invertible data in a controlled manner. With add' defined, fib can be written as follows.

```
sig  fib : Nat ↔ (Nat, Nat)                  sig is11 : Nat → Bool
def• fib n =                                 def is11 n =
  case• n of                                   case n of
    Z   → (S Z, S Z)   with is11                 (S Z, S Z) → True
    S n → let• (y, x) = fib ◇ n in               _          → False
            add' ◇ (x, y)
              with not ∘ is11
```

This example is defined by invertible pattern matching (**case•**), a construct inherited from previous languages like Janus [35,53] and Ψ-Lisp [7]. When branching on the input to a bijection (as opposed to a fixed parameter), postconditions marked by the keyword **with** ensure that the execution can determine which branch to take in the backward direction. Each postcondition is a boolean function that must return True for any result of its branch and False for any result of the branches below it (this is checked at runtime following the symmetric first-match policy [54]). The backward evaluation tests each condition in turn, selecting the first branch whose condition is true. Here, $is11$ is used to distinguish the base case where the output is (S Z, S Z).

The inverse behavior of fib computes n given a pair (F_n, F_{n+1}). Specifically, by computing $F_{n+1} - F_n$, we obtain F_{n-1}, and repeating the process until we reach the start of the sequence lets us deduce the index of the initial pair. Kalpis runs fib as below.

```
-- (F3, F4) = (3, 5)                          -- (Fn, Fn+1) = (3, 5)   ⇒   n = 3
> fib ◇ S (S (S Z))                           > fib† ◇ (S (S (S Z)), S (S (S (S (S Z)))))
(S (S (S Z)), S (S (S (S (S Z)))))            S (S (S Z))
```

Again, fib is non-total: running it backwards on a pair not constituting two consecutive Fibonacci numbers will cause the computation to fail.

Viewed as an operation on parameterized bijections, pin lets part of an invertible input be shifted to the parameter position if a copy is returned in the end. Formally, we have $pin(f)_n(x, y) = (x, f_{(n,x)}(y))$; in our example, $f_{(n,x)}$ corresponds to addition by x, ignoring a trivial n representing variables captured in the pin form.

2.3 Partially Invertible Composition

We now return to the example of the introduction, *autokey*. It can be defined in Kalpis as follows:

```
sig   $autokey : \mathsf{Char} \to [\mathsf{Char}] \leftrightarrow [\mathsf{Char}]$
def•  $autokey\ k\ xs =$
  case• $xs$ of
    $[]$       $\to []$
    $(h : t)$ $\to$ let• $(h, r) = pin\ autokey \diamond (h, t)$ in
                   $(shift\ (chrToInt\ k) \diamond h) : r$
```

The structure is very similar to the unidirectional version in Section 1, but uses the invertible branching and pinning constructs explained previously. We assume primitives *chrToInt* : $\mathsf{Char} \to \mathsf{Int}$ and *shift* : $\mathsf{Int} \to \mathsf{Char} \leftrightarrow \mathsf{Char}$ for computing and performing the cyclical shifts, respectively. We omit the **with**-conditions of the invertible match by convention, as the syntactically distinct branch bodies can act as patterns to guide backward branching.

This example features our third core construct, *partially invertible composition*. This simply refers to the fact that we can modify the parameter of a bijection unidirectionally, as in *shift* (*chrToInt k*) $\diamond$ *h*. In this case, the (irreversible) function *chrToInt* is applied to *k* inside the (invertible) call to *shift*. In other words, the *parameter* part of an invertible computation is allowed to depend freely on unidirectional computations, greatly enhancing the flexibility when programming. The reason we call it *composition* is because from the perspective of parameterized bijections, this corresponds to the composition of a parameterized bijection f with an (arbitrary) function g on the parameter part, *i.e.,* $(f \circ g)_n(x) = f_{g(n)}(x)$. In our example, we have f corresponding to *shift* and g corresponding to *chrToInt*.

The example also further highlights the utility of *pin*. As noted in the introduction, *autokey* is tricky to express since each character in the invertible output depends unidirectionally on the preceding character in the corresponding input. Similar patterns also appear in more advanced examples; for instance, consider an adaptive compression method where each character in the input must be treated invertibly, and yet also be used as part of the (unidirectional) compression table. *pin* enables this sort of dependency in a safe way, letting us use h in the recursive call to *autokey* and returning a copy to use in the output.

Again, Kalpis lets us execute *autokey* in either direction, and guarantees that the two are inverses.

```
> autokey 'F' ◇ "HELLO"              > (autokey 'F')† ◇ "CXHAD"
"CXHAD"                              "HELLO"
```

2.4 Abstraction and Application of Invertible Computations

Our final core construct of partially invertible programming is the ability to abstract and apply invertible computations. Although the examples we have seen so far have defined (partially) invertible computations using the **def**$^\bullet$ keyword in a style close to traditional invertible languages, KALPIS actually features bijections as first-class values and supports proper higher-order programming. Bijections can be constructed with an invertible λ-form $\lambda^\bullet x.e$ analogous to that typical for ordinary functions, and the form **def**$^\bullet$ $f\ x_1\ x_2\ \ldots\ x_n = e$ is simply syntactic sugar for $f = \lambda x_1.\lambda x_2 \ldots \lambda^\bullet x_n.\ e$. To our knowledge, only SPARCL [39, 40] shares this feature, with most invertible languages being limited to first-order computation.

For example, we are able to define multiple variants of the typical *map* function for lists in KALPIS.

```
sig map : (a → b) → [a] → [b]          sig  mapBij : (a ↔ b) → [a] ↔ [b]
def map f l =                          def• mapBij f l =
  case xs of                             case• xs of
    []    → []                             []    → []
    h : t → f h : map f t                  h : t → (f ◇ h) : (mapBij f ◇ t)
```

Here, *map* is defined as usual, and maps a function over each element of a list, while *mapBij* makes use of the language's invertible constructs, taking a bijection argument to produce a bijection on lists. For example, using *mapBij*, the Caesar cipher (which shifts each character in the input a fixed number of steps) can be defined with a one-liner, as below to the left.

```
sig caesar : Char → [Char] ↔ [Char]      sig vig : [Char] → [Char] ↔ [Char]
def caesar k = mapBij (shift k)          def vig ks = apBij (map shift ks)
```

The function on the right, *vig* (from Vigenère), takes a list of keys, shifting each character in the input using the corresponding key—the definition relies on $apBij : [a \leftrightarrow b] \to [a] \leftrightarrow [b]$ to apply a list of bijections pointwise to a list of inputs (assuming the two have equal lengths). The latter example demonstrates that bijections can even occur inside data structures such as lists.

Some restrictions must be observed when dealing with higher-order computation in KALPIS. The language distinguishes between unidirectional and invertible terms, and carefully controls the interaction between the two. The restrictions mean that the *invertible fragment* of the language is essentially first-order; a formal account is given in Section 3.

Viewed from the perspective of parameterized bijections, abstraction corresponds to forming the function $n \mapsto f_n$, witnessing that each choice of parameter n induces a bijection f_n which can be treated as a standalone value. On the other hand, application of a bijection α corresponds to forming the parameterized bijection $app_\alpha(x) = \alpha(x)$, where the parameter determining the bijection is α itself.

This concludes Section 2; for more programming examples in KALPIS, we refer to the prototype implementation,[8] which contains a number of nontrivial programs, including implementations of Huffman coding and sliding-window compression.

[8] https://git.sr.ht/~aathn/kalpis

3 The Kalpis Core System

In this section, we formally define the Kalpis core system and state the essential metatheoretic properties. A salient feature of the system is the clear separation between unidirectional and invertible terms: we have two main syntactic categories, two typing relations, and three evaluation relations (one for unidirectional terms, and one in each direction for invertible terms). The unidirectional terms are a conservative extension of a standard simply-typed call-by-value λ-calculus, and the invertible terms add support for (partially) invertible computation.

After introducing the syntax and reviewing some examples, Sections 3.4 and 3.5 give a formal semantics which suggests an interpretation of Kalpis terms as parameterized bijections. This view is made precise in Sections 4 and 5, which define a translation from Kalpis into the arrow language rrArr, enabling a locally invertible interpretation.

3.1 Syntax

The syntax of Kalpis core is given below, where u denotes unidirectional terms, r denotes invertible terms, and p denotes patterns. The vector notation $\overline{t}$ denotes an ordered sequence of elements t_i, whose length we will refer to by $|\overline{t}|$.

$$\begin{aligned}
u &::= x \mid \lambda x.u \mid u_1\ u_2 \mid \lambda^\bullet x.r \mid u_1 \diamond u_2 \mid \mathsf{C}\ \overline{u} \mid \textbf{case}\ u_0\ \textbf{of}\ \{\overline{p \to u}\} \\
r &::= x \mid u \diamond r \mid u^\dagger \diamond r \mid \mathit{pin}\ u \diamond r \\
&\mid\ \ \mathsf{C}\ \overline{r} \mid \textbf{case}\ u\ \textbf{of}\ \{\overline{p \to r}\} \mid \textbf{case}^\bullet\ r_0\ \textbf{of}\ \{\overline{p \to r\ \textbf{with}\ u}\} \\
p &::= \mathsf{C}\ \overline{x}
\end{aligned}$$

The syntax of unidirectional terms include the standard cases for variables, abstraction and application, along with data constructors and pattern matching. In addition, there is the invertible abstraction $\lambda^\bullet x.r$ and application $u_1 \diamond u_2$ explained in the previous section. Note that while the body r is an invertible term, the abstraction itself is unidirectional.

The syntax of invertible terms resembles a first-order functional language, but with a couple of key additions. We have bijection application $u \diamond r$, where the bijection is unidirectional whereas the argument is invertible. We also have fully applied versions of the $(\cdot)^\dagger$ and *pin* operators explained in the previous section (this is without loss of generality, as *e.g.,* the higher-order version of *pin* can be recovered as $\lambda f.\lambda^\bullet x.\ \mathit{pin}\ f \diamond x$). Partially invertible branching is represented by the **case** form, whose scrutinee u is unidirectional. The **case**$^\bullet$ form deconstructs an invertible term, and has a **with**-condition for invertible branching, following Janus [35, 53] and Ψ-Lisp [7]. The core constructs of the previous section are all featured explicitly in the syntax, except for partially invertible composition, which is implicitly performed whenever a unidirectional term u occurs in an invertible context.

3.2 Types

Next, we define the types of Kalpis core.

$$A, B ::= \mathsf{T}\ \overline{B} \mid A \to B \mid A \leftrightarrow B \mid X$$

The types include constructors $\mathsf{T}\ \overline{B}$, functions $A \to B$, bijections $A \leftrightarrow B$ and type variables X. The types are conventional with the exception of invertible computations $A \leftrightarrow B$; this simplicity is a design feature of KALPIS. With each type constructor T we associate an arity k and a set of constructors C with signatures $\mathsf{C} : A_1 \to A_2 \to \cdots \to A_n \to \mathsf{T}\ \overline{B}$, where $|\overline{B}| = k$. We will assume the type constructors include at least the unit $\mathsf{1}$, products $\otimes$, and sums $\oplus$ with constructors

$$() : \mathsf{1} \qquad (-,-) : A \to B \to A \otimes B \qquad \mathsf{InL} : A \to A \oplus B \qquad \mathsf{InR} : B \to A \oplus B$$

for any A, B. We use Bool as a shorthand for $\mathsf{1} \oplus \mathsf{1}$, and True, False as shorthands for $\mathsf{InL}\ (), \mathsf{InR}\ ()$, respectively.

Types can be (mutually) recursive via constructors; for example, the type Nat has constructors $\mathsf{Z} : \mathsf{Nat}$ and $\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$. In general, for any fixed A, the recursive type $\mu X.A$ can be represented with a nullary type constructor Rec_A, with constructor

$$\mathsf{Roll} : A[\mathsf{Rec}_A/X] \to \mathsf{Rec}_A.$$

For instance, $\mathsf{Rec}_{\mathsf{1}\oplus X}$ has constructor $\mathsf{Roll} : \mathsf{1} \oplus \mathsf{Rec}_{\mathsf{1}\oplus X} \to \mathsf{Rec}_{\mathsf{1}\oplus X}$, making it isomorphic to Nat. Technically, we consider a variable X implicitly bound in the annotation to Rec, and assume all other types are closed.

3.3 Correspondence to the Surface Language

The correspondence between the core syntax and the examples of Section 2 should be clear. For instance, the examples of addition and Fibonacci number calculation can be written as follows:

```
add ≜ fix (λadd′.λn.λ•m.
        case n of
          Z    → m
          S n′ → S (add′ n′ ◇ m)))
```

```
fib ≜ fixBij (λfib′.λ•n.
        case• n of
          Z    → (S Z, S Z) with is11
          S n′ → case• fib′ ◇ n′ of
                   (x, y) → pin add ◇ (y, x))
                             with λ_. True
                 with not ∘ is11)
```

Here, *add* is a unidirectional term defined using a fixpoint operator *fix*, and the structure is similar to the version presented in Section 2.1. The function *fib* is similarly defined, but uses the fixpoint operator *fixBij* instead of *fix*, which works for bijections instead of functions. We omit the definition of $\mathit{is11} : \mathsf{Nat} \otimes \mathsf{Nat} \to \mathsf{Bool}$ in the interest of space. The term *fixBij* (and analogously *fix*) is defined as below, making use of the language's recursive types.

$$\mathit{fixBij} \triangleq \lambda f.\ (\lambda g.\ g\ (\mathsf{Roll}\ g))\ (\lambda x.\lambda^\bullet a.\ f\ ((\mathbf{case}\ x\ \mathbf{of}\ \mathsf{Roll}\ y \to y)\ x) \diamond a)$$

The type system we define in the next section will assign these terms the following types as expected.

$$\begin{array}{llll} \mathit{add} & : \mathsf{Nat} \to \mathsf{Nat} \leftrightarrow \mathsf{Nat} & \mathit{fix} & : ((A \to B) \to A \to B) \to A \to B \\ \mathit{fib} & : \mathsf{Nat} \leftrightarrow \mathsf{Nat} \otimes \mathsf{Nat} & \mathit{fixBij} & : ((A \leftrightarrow B) \to A \leftrightarrow B) \to A \leftrightarrow B \end{array}$$

3.4 Type System

Figure 1 shows the typing rules for unidirectional ($\Gamma \vdash u : A$) and invertible ($\Gamma; \Theta \vdash r : A$) terms. The latter relation uses two contexts Γ and Θ; intuitively, Γ contains variables for unidirectional data, which may be discarded or duplicated freely, whereas Θ contains variables for data that must be treated in an invertible way. This use of a dual context system [13] is inspired by previous work such as CoreFun [25] and Sparcl [39, 40]. Formally, we define the typing contexts as $\Gamma, \Theta ::= \varepsilon \mid \Gamma, x : A$, and assume names x are unique within a context. We let Γ_1, Γ_2 denote the concatenation of two contexts.

The rules for $\Gamma \vdash u : A$ are mostly straightforward. T-Abs$^\bullet$ pushes the parameter x of $\lambda^\bullet x.r$ into Θ instead of Γ to ensure that the variable is used in an invertible way in r, and T-Run gives a rule for bijection application analogous to T-App. In the Case rules, we implicitly require that patterns are disjoint and exhaustive.

In the rules for $\Gamma; \Theta \vdash r : A$, the variables in the Θ environments must be used exactly once to ensure invertibility. Hence, we need to separate Θ into, *e.g.*, $\Theta = \Theta_1 \uplus \Theta_2$ for typing subterms, where $\uplus$ is used analogously to a linear type system (see, *e.g.,* [9]). The rules follow the intuition that r denotes a bijection between Θ and A parameterized by Γ. This highlights the difference between the pattern matching rules, T-UCase and T-RCase: the bound variables Γ_i in the former are parameters for the bijection that r_i defines, while in the latter, the variables Θ_i are part of the inputs of r_i, so that **case**$^\bullet$ performs a composition of two invertible computations.

As stated in Section 2.4, there are some restrictions on how unidirectional and invertible terms can interact. Note that the unidirectional subterms occurring in the invertible typing rules are only typed using Γ, and not Θ. For instance, since the left-hand side in rule T-RApp is unidirectional, it cannot depend directly on invertible variables, ruling out terms like $\lambda^\bullet x.(x \diamond \mathsf{True})$. This is a natural restriction, as we cannot generally deduce which function was used to produce some given result. Conversely, there is no rule for directly accessing Γ from the invertible typing relation; instead, unidirectional data can only affect the computation through rules like T-UCase and T-RApp. Both λ-forms are unidirectional, meaning they can neither capture invertible variables nor be returned from an invertible computation. In this sense, the invertible fragment of the language is first-order.

We note that there are no particular restrictions on unidirectional terms, and the approach presented could be used to augment any standard functional language with invertible computations $\lambda^\bullet x.r$ and $u_1 \diamond u_2$. The prototype implementation further adds **let**-polymorphism as an orthogonal extension.

3.5 Operational Semantics

We first define the set of values as below.

$$v ::= \mathsf{C}\, \overline{v} \mid \langle \lambda x.u, \gamma \rangle \mid \langle \lambda^\bullet x.r, \gamma \rangle$$

Typing Rules for Unidirectional Terms $\boxed{\Gamma \vdash u : A}$ **and Patterns** $\boxed{\Gamma \vdash p : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-UVar} \qquad \frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x.u : A \to B}\ \text{T-Abs} \qquad \frac{\Gamma \vdash u_1 : A \to B \quad \Gamma \vdash u_2 : A}{\Gamma \vdash u_1\ u_2 : B}\ \text{T-App}$$

$$\frac{\Gamma; x : A \vdash r : B}{\Gamma \vdash \lambda^{\bullet} x.r : A \leftrightarrow B}\ \text{T-Abs}^{\bullet} \qquad \frac{\Gamma \vdash u_1 : A \leftrightarrow B \quad \Gamma \vdash u_2 : A}{\Gamma \vdash u_1 \diamond u_2 : B}\ \text{T-Run}$$

$$\frac{|\overline{u}| = |\overline{A}| \quad \mathsf{C} : \overline{A} \to \mathsf{T}\,\overline{B} \quad \{\Gamma \vdash u_i : A_i\}_i}{\Gamma \vdash \mathsf{C}\,\overline{u} : \mathsf{T}\,\overline{B}}\ \text{T-Con}$$

$$\frac{\Gamma \vdash u_0 : A \quad \{\Gamma_i \vdash p_i : A \quad \Gamma, \Gamma_i \vdash u_i : B\}_i}{\Gamma \vdash \textbf{case}\ u_0\ \textbf{of}\ \{\overline{p \to u}\} : B}\ \text{T-Case} \qquad \frac{|\overline{x}| = |\overline{A}| \quad \mathsf{C} : \overline{A} \to \mathsf{T}\,\overline{B}}{\overline{x : A} \vdash \mathsf{C}\,\overline{x} : \mathsf{T}\,\overline{B}}\ \text{T-Pat}$$

Typing Rules for Invertible Terms $\boxed{\Gamma; \Theta \vdash r : A}$

$$\frac{}{\Gamma; x : A \vdash x : A}\ \text{T-RVar} \qquad \frac{\Gamma \vdash u : A \leftrightarrow B \quad \Gamma; \Theta \vdash r : A}{\Gamma; \Theta \vdash u \diamond r : B}\ \text{T-RApp}$$

$$\frac{\Gamma \vdash u : A \leftrightarrow B \quad \Gamma; \Theta \vdash r : B}{\Gamma; \Theta \vdash u^{\dagger} \diamond r : A}\ \text{T-Inv} \qquad \frac{\Gamma \vdash u : C \to A \leftrightarrow B \quad \Gamma; \Theta \vdash r : C \otimes A}{\Gamma; \Theta \vdash \mathit{pin}\ u \diamond r : C \otimes B}\ \text{T-Pin}$$

$$\frac{|\overline{\Theta}| = |\overline{r}| = |\overline{A}| \quad \mathsf{C} : \overline{A} \to \mathsf{T}\,\overline{B} \quad \{\Gamma; \Theta_i \vdash r_i : A_i\}_i}{\Gamma; \uplus\overline{\Theta} \vdash \mathsf{C}\,\overline{r} : \mathsf{T}\,\overline{B}}\ \text{T-RCon}$$

$$\frac{\Gamma \vdash u : A \quad \{\Gamma_i \vdash p_i : A \quad \Gamma, \Gamma_i; \Theta \vdash u_i : B\}_i}{\Gamma; \Theta \vdash \textbf{case}\ u\ \textbf{of}\ \{\overline{p \to r}\} : B}\ \text{T-UCase}$$

$$\frac{\Gamma; \Theta \vdash r_0 : A \quad \big\{\Theta_i \vdash p_i : A \quad \Gamma; \Theta' \uplus \Theta_i \vdash r_i : B \quad \Gamma \vdash u_i : B \to \mathsf{Bool}\big\}_i}{\Gamma; \Theta \uplus \Theta' \vdash \textbf{case}^{\bullet}\ r_0\ \textbf{of}\ \{\overline{p \to r\ \textbf{with}\ u}\} : B}\ \text{T-RCase}$$

Fig. 1. The type system of Kalpis core: $\overline{A} \to B$ means $A_1 \to \cdots \to A_{|\overline{A}|} \to B$.

Here, γ is a value environment, *i.e.*, a mapping from variables to their values. Formally, we define $\gamma, \theta ::= \emptyset \mid \gamma, x \mapsto v$, with γ and θ corresponding to Γ and Θ. We use the disjoint union $\theta_1 \uplus \theta_2$ to concatenate two environments θ_1 and θ_2, which is defined only when $\mathsf{dom}(\theta_1)$ and $\mathsf{dom}(\theta_2)$ are disjoint. The values include constructors and two closure forms $\langle \lambda x.u, \gamma \rangle$ and $\langle \lambda^{\bullet} x.r, \gamma \rangle$, corresponding to unidirectional and invertible computations. We type the values in analogy with the terms, with the rules for closures as follows:

$$\frac{\gamma : \Gamma \quad \Gamma, x : A \vdash u : B}{\langle \lambda x.u, \gamma \rangle : A \to B} \qquad \frac{\gamma : \Gamma \quad \Gamma; x : A \vdash r : B}{\langle \lambda^{\bullet} x.r, \gamma \rangle : A \leftrightarrow B}$$

Here, we write $\gamma : \Gamma$ to mean that $\mathsf{dom}(\gamma) = \mathsf{dom}(\Gamma)$ and $\gamma(x) : \Gamma(x)$ for all $x \in \mathsf{dom}(\Gamma)$. For p a pattern, we write $p\gamma$ to denote the value obtained by applying the substitution γ to p's variables. In addition, we use the shorthand $\widehat{i = j} \triangleq \begin{cases} \mathsf{True} & \text{if } i = j \\ \mathsf{False} & \text{otherwise} \end{cases}$.

We now present in Figure 2 the operational semantics of Kalpis core, which consists of three evaluation relations: unidirectional, forward, and backward. The unidirectional evaluation relation $\gamma \vdash u \Downarrow v$ reads that under γ term u evaluates to value v, as usual. In contrast, the forward and backward evaluation relations define a bijection. The former relation $\gamma; \theta \vdash r \Rightarrow v$ reads that under γ the forward evaluation of r maps θ to v, and the latter relation $\gamma; v \vdash r \Leftarrow \theta$ reads that under γ the backward evaluation of r maps v to θ. As one can see, γ serves as parameter for this bijection that defines a one-to-one correspondence between θ and v. Due to the space limitations, we omitted the rules for backward evaluation, as they are completely symmetric to forward evaluation. That is, for each rule of the forward evaluation, the corresponding backward rule is obtained by swapping each occurrence $\gamma; \theta \vdash r \Rightarrow v$ with $\gamma; v \vdash r \Leftarrow \theta$, and vice versa. Crucially, the evaluation relations are mutually dependent, and when a unidirectional term is embedded in an invertible computation, the unidirectional evaluation will be invoked to evaluate the term in the same way regardless of whether executing forwards or backwards.

We encourage the reader to study the rules for partially invertible **case** and invertible **case**$^\bullet$ especially. The former branches based on a unidirectional term, which is evaluated first regardless of the direction of execution. The latter branches based on an invertible term, which is evaluated first in the forward direction but last in the backward direction. In the backward direction, the **with**-conditions $\overline{u}$ are instead evaluated first; the condition $\widehat{i = j}$ for $j \leq i$ encodes the branch selection and the runtime check of postconditions mentioned previously.

There is a subtlety in the backward evaluation rule for constructors $\mathsf{C}\ \overline{r}$, where the same C occurs both in the term $\mathsf{C}\ \overline{r}$ and the input $\mathsf{C}\ \overline{v}$, meaning that evaluation fails if the value does not match the constructor C. This corresponds to, *e.g.,* the term $(\lambda^\bullet x.\ \mathsf{S}\ x)^\dagger \diamond \mathsf{Z}$ failing as it tries to subtract one from zero.

3.6 Metatheory

In this section, we briefly state the essential properties of the core system. The propositions in this section have been formalized mechanically, by implementing and reasoning about a definitional interpreter [46] in Agda. The implementation follows the presentation of the paper closely, but uses intrinsically-typed terms and nameless variables, and relies on the sized delay monad [1, 11].

Theorem 1 (Subject reduction).
- *If $\Gamma \vdash u : A$, $\gamma : \Gamma$ and $\gamma \vdash u \Downarrow v$, then $v : A$.*
- *If $\Gamma; \Theta \vdash r : A$, $\gamma : \Gamma$, $\theta : \Theta$ and $\gamma; \theta \vdash r \Rightarrow v$, then $v : A$.*
- *If $\Gamma; \Theta \vdash r : A$, $\gamma : \Gamma$, $v : A$ and $\gamma; v \vdash r \Leftarrow \theta$, then $\theta : \Theta$.*

Proof. Directly from the existence and type of the definitional interpreter in Agda.

Theorem 2 (Invertibility). *If $\Gamma; \Theta \vdash r : A$, $\gamma : \Gamma$, $\theta : \Theta$ and $v : A$, then*

$$\gamma; \theta \vdash r \Rightarrow v \quad \textit{if and only if} \quad \gamma; v \vdash r \Leftarrow \theta.$$

Proof. By simultaneous induction on the term r and the step count of evaluation; simple induction on the term r is not enough as the language has general recursion. The proof is otherwise straightforward, since the evaluation relations are completely symmetric.

Unidirectional Evaluation $\boxed{\gamma \vdash u \Downarrow v}$

$$\frac{\gamma(x) = v}{\gamma \vdash x \Downarrow v} \qquad \frac{}{\gamma \vdash \lambda x.u \Downarrow \langle \lambda x.u, \gamma\rangle} \qquad \frac{}{\gamma \vdash \lambda^\bullet x.r \Downarrow \langle \lambda^\bullet x.r, \gamma\rangle}$$

$$\frac{\gamma \vdash u_1 \Downarrow \langle \lambda x.u, \gamma'\rangle \quad \gamma \vdash u_2 \Downarrow v_2 \quad \gamma', x \mapsto v_2 \vdash u \Downarrow v}{\gamma \vdash u_1\ u_2 \Downarrow v} \qquad \frac{\{\gamma \vdash u_i \Downarrow v_i\}_i}{\gamma \vdash \mathsf{C}\ \overline{u} \Downarrow \mathsf{C}\ \overline{v}}$$

$$\frac{\gamma \vdash u_1 \Downarrow \langle \lambda^\bullet x.r, \gamma'\rangle \quad \gamma \vdash u_2 \Downarrow v_2 \quad \gamma'; x \mapsto v_2 \vdash r \Rightarrow v}{\gamma \vdash u_1 \diamond u_2 \Downarrow v} \qquad \frac{\gamma \vdash u_0 \Downarrow p_i\gamma_i \quad \gamma, \gamma_i \vdash u_i \Downarrow v'}{\gamma \vdash \textbf{case}\ u_0\ \textbf{of}\ \{\overline{p \to u}\} \Downarrow v'}$$

Forward (and Backward) Evaluation $\boxed{\gamma; \theta \vdash r \Rightarrow v}$ $\left(\boxed{\gamma; v \vdash r \Leftarrow \theta}\right)$

$$\frac{\gamma \vdash u \Downarrow \langle \lambda^\bullet x.r', \gamma'\rangle \quad \gamma; \theta \vdash r \Rightarrow v \quad \gamma'; x \mapsto v \vdash r' \Rightarrow v'}{\gamma; \theta \vdash u \diamond r \Rightarrow v'} \qquad \frac{}{\gamma; (x \mapsto v) \vdash x \Rightarrow v}$$

$$\frac{\gamma \vdash u \Downarrow \langle \lambda^\bullet x.r', \gamma'\rangle \quad \gamma; \theta \vdash r \Rightarrow v \quad \gamma'; v \vdash r' \Leftarrow (x \mapsto v')}{\gamma; \theta \vdash u^\dagger \diamond r \Rightarrow v'} \qquad \frac{\{\gamma; \theta_i \vdash r_i \Rightarrow v_i\}_i}{\gamma; \uplus\overline{\theta} \vdash \mathsf{C}\ \overline{r} \Rightarrow \mathsf{C}\ \overline{v}}$$

$$\frac{\begin{array}{c}\gamma \vdash u \Downarrow \langle \lambda x.u', \gamma'\rangle \quad \gamma; \theta \vdash r \Rightarrow (v_1, v_2) \\ \gamma', x \mapsto v_1 \vdash u' \Downarrow \langle \lambda^\bullet y.r', \gamma''\rangle \quad \gamma''; y \mapsto v_2 \vdash r' \Rightarrow v_3\end{array}}{\gamma; \theta \vdash \mathit{pin}\ u \diamond r \Rightarrow (v_1, v_3)} \qquad \frac{\gamma \vdash u \Downarrow p_i\gamma_i \quad \gamma, \gamma_i; \theta \vdash r_i \Rightarrow v'}{\gamma; \theta \vdash \textbf{case}\ u\ \textbf{of}\ \{\overline{p \to r}\} \Rightarrow v'}$$

$$\frac{\gamma; \theta \vdash r_0 \Rightarrow p_i\theta_i \quad \gamma; \theta', \theta_i \vdash r_i \Rightarrow v' \quad \left\{\gamma \vdash u_j \Downarrow \langle \lambda x.u'_j, \gamma_j\rangle \quad \gamma_j, x \mapsto v' \vdash u'_j \Downarrow \widehat{i = j}\right\}_{j \le i}}{\gamma; \theta \uplus \theta' \vdash \textbf{case}^\bullet\ r_0\ \textbf{of}\ \{\overline{p \to r\ \textbf{with}\ u}\} \Rightarrow v'}$$

Fig. 2. The operational semantics of KALPIS core. Rules for the backward evaluation are omitted in the interest of space, but can be derived as explained in the text.

Remark on Progress. We have chosen to give the semantics in a big-step style in this paper. This choice was made both because the invertibility property is more natural to state about a big-step semantics, which relates input to output directly, and to make the step to a denotational semantics smaller—as mentioned, the evaluation relations suggest an interpretation of invertible terms as parameterized bijections.

Thus, the progress property typically proven for a small-step semantics, meaning that evaluation never gets "stuck" given a valid input (see, *e.g.,* [45]), is not direct to state in our case. However, we get a similar guarantee from the implementation in Agda, whose type checker asserts that no uncontrolled run-time errors are possible. Indeed, the only errors that can occur during evaluation are those caused by imprecise **with**-conditions or mismatched constructors.

4 Arrows for Partial and Local Invertibility

While the core system of KALPIS presented in the previous section is simple and illuminating, it only offers an operational understanding of the language. Furthermore, it depends on a unidirectional evaluation, which does not fit in a locally invertible setting. We want to get at the essence of partially invertible programming, and show that partial and local invertibility can be reconciled, which is the focus of this section.

Syntax

$$
\begin{aligned}
A, B &::= 1 \mid A \oplus B \mid A \otimes B \mid \mu X.A \\
\tau &::= A \leftrightharpoons B \mid A \rightsquigarrow B \mid C \cdot A \leftrightsquigarrow B \\
\mu &::= \mathit{arr}_{\mathrm{u}}\ c \mid \mu_1 \ggg_{\mathrm{u}} \mu_2 \mid \mathit{first}_{\mathrm{u}}\ \mu \mid \mathit{left}_{\mathrm{u}}\ \mu \mid \mathit{clone} \mid \mathit{run}\ \alpha \\
\alpha &::= \mathit{arr}_{\mathrm{r}}\ c \mid \alpha_1 \ggg_{\mathrm{r}} \alpha_2 \mid \mathit{first}_{\mathrm{r}}\ \alpha \mid \mathit{left}_{\mathrm{r}}\ \alpha \mid \alpha^{\dagger} \mid \mathit{case!}\ \alpha_1\ \alpha_2 \mid \mathit{pin}\ \alpha \mid \mu \ggg!\ \alpha
\end{aligned}
$$

Typing Rules for Arrows $\boxed{\mu : A \rightsquigarrow B}$ **and** $\boxed{\alpha : C \cdot A \leftrightsquigarrow B}$

$$
\frac{c : A \leftrightharpoons B}{\mathit{arr}_{\mathrm{u}}\ c : A \rightsquigarrow B}
\qquad
\frac{\mu_1 : A \rightsquigarrow B \quad \mu_2 : B \rightsquigarrow C}{\mu_1 \ggg_{\mathrm{u}} \mu_2 : A \rightsquigarrow C}
\qquad
\frac{\mu : A \rightsquigarrow B}{\mathit{first}_{\mathrm{u}}\ \mu : A \otimes C \rightsquigarrow B \otimes C}
$$

$$
\frac{\mu : A \rightsquigarrow B}{\mathit{left}_{\mathrm{u}}\ \mu : A \oplus C \rightsquigarrow B \oplus C}
\qquad
\frac{}{\mathit{clone} : A \rightsquigarrow A \otimes A}
\qquad
\frac{\alpha : C \cdot A \leftrightsquigarrow B}{\mathit{run}\ \alpha : C \otimes A \rightsquigarrow B}
$$

$$
\frac{c : A \leftrightharpoons B}{\mathit{arr}_{\mathrm{r}}\ c : C \cdot A \leftrightsquigarrow B}
\qquad
\frac{\alpha_1 : D \cdot A \leftrightsquigarrow B \quad \alpha_2 : D \cdot B \leftrightsquigarrow C}{\alpha_1 \ggg_{\mathrm{r}} \alpha_2 : D \cdot A \leftrightsquigarrow C}
\qquad
\frac{\alpha : D \cdot A \leftrightsquigarrow B}{\mathit{first}_{\mathrm{r}}\ \alpha : D \cdot (A \otimes C) \leftrightsquigarrow B \otimes C}
$$

$$
\frac{\alpha : D \cdot A \leftrightsquigarrow B}{\mathit{left}_{\mathrm{r}}\ \alpha : D \cdot (A \oplus C) \leftrightsquigarrow B \oplus C}
\qquad
\frac{\alpha : C \cdot A \leftrightsquigarrow B}{\alpha^{\dagger} : C \cdot B \leftrightsquigarrow A}
\qquad
\frac{\alpha_1 : C \cdot A \rightsquigarrow B \quad \alpha_2 : D \cdot A \rightsquigarrow B}{\mathit{case!}\ \alpha_1\ \alpha_2 : (C \oplus D) \cdot A \leftrightsquigarrow B}
$$

$$
\frac{\mu : C \rightsquigarrow D \quad \alpha : D \cdot A \leftrightsquigarrow B}{\mu \ggg!\ \alpha : C \cdot A \leftrightsquigarrow B}
\qquad
\frac{\alpha : (C \otimes D) \cdot A \leftrightsquigarrow B}{\mathit{pin}\ \alpha : C \cdot (D \otimes A) \leftrightsquigarrow D \otimes B}
$$

Fig. 3. The syntax and types of rrArr: A and B denote base types, τ denotes combinator types, c denotes bijections, μ denotes unidirectional arrow combinators and α denotes invertible arrow combinators.

In what follows, we define rrArr, a low-level language based on arrow combinators, intended to capture the essence of partially invertible computation. The operations of rrArr directly correspond to the core constructs of Section 2, and have an immediate interpretation in terms of abstract functions and parameterized bijections. What is more, we show that they have an alternative, *compositional* and *locally invertible* interpretation using an idea similar to the reader monad in unidirectional computation (based on the irreversibility effect [26] and the reversible reader [23]). This property is not obvious for Kalpis, not to mention earlier work such as Sparcl [39, 40].

We begin by explaining the syntax and semantics of a first-order fragment of rrArr, before proceeding to give its locally invertible intrepretation. We then extend this fragment to match the full expressiveness of Kalpis in Section 4.5 with operations for higher-order computation. In Section 5, we top it all off by giving a formal translation from Kalpis core to rrArr.

4.1 Syntax and Type System of rrArr

Figure 3 shows the syntax and type system of rrArr (where base bijections c of type $A \leftrightharpoons B$ are kept abstract). The language involves unidirectional (μ) and invertible (α) terms, similarly to Kalpis. Both kinds of terms form arrows over bijections, through the combinators *arr*, $\ggg$, and *first*.

The former arrow, denoted by $\mu : A \rightsquigarrow B$, intuitively represents an ordinary function; $\mathit{arr}_{\mathrm{u}}\ c$ extracts the forward semantics of a bijection c, $\mu_1 \ggg_{\mathrm{u}} \mu_2$

composes two functions μ_1 and μ_2, and $\mathit{first}_{\mathrm{u}}\ \mu$ simply applies μ to the first component of the input. The unidirectional arrows also feature $\mathit{left}_{\mathrm{u}}$, the sum counterpart of *first*, and allow copying data through *clone*.

The latter arrow, denoted by $\alpha : C \cdot A \leftrightsquigarrow B$, represents bijections between A and B parameterized by C; $\mathit{arr}_{\mathrm{r}}\ c$ constructs a parameterized bijection that behaves as the bijection c ignoring any parameter, $\alpha_1 \ggg_{\mathrm{r}} \alpha_2$ composes the two bijections obtained by passing the parameter to both α_1 and α_2, and $\mathit{first}_{\mathrm{r}}\ \alpha$ applies the bijection determined by α to the first component of the input. These arrows also support $\mathit{left}_{\mathrm{r}}$, and form an inverse arrow [23] through a dagger operator $\alpha^{\dagger}$, that undoes α and its effect.

What is special in RRARR is the communication between the two arrows through *case*!, *pin*, $\gg!$, and *run*, where the former three directly correspond to the core constructs of Section 2. The term *case*! $\alpha_1\ \alpha_2$ performs partially invertible branching, running α_1 or α_2 depending on the value of its parameter. The term *pin* α corresponds to the pinning construct; in RRARR, this operation moves part of the input (D) into the parameter ($C \otimes D$) of α. The term $\mu \gg! \alpha$ represents partially invertible composition of the function μ with the parameterized bijection α. Finally, the operator *run* allows converting a parameterized bijection $C \cdot A \leftrightsquigarrow B$ to a function $C \otimes A \rightsquigarrow B$ by extracting its forward semantics. This can be seen as a special case of applying invertible computations (in a unidirectional context); the treatment of abstraction and application supporting higher-order computation is left for Section 4.5, as it requires a slight extension.

It is worth noting that invertible arrows are inherently allowed to ignore their parameter (through $\mathit{arr}_{\mathrm{r}}$), a fact that can be used to derive the crucial erasure operation in unidirectional arrows. In particular, supposing $\mathit{id} : A \leftrightharpoons A$, we get the term $\mathit{run}\ (\mathit{arr}_{\mathrm{r}}\ \mathit{id}) : C \otimes 1 \rightsquigarrow 1$, which ignores any input C to return $()$.

4.2 Semantics of RRARR

We now formalize the intuitive interpretation through the semantics presented in Figure 4. We define a base set of values containing unit, pairs, and tagged values, which we type in the conventional way. Recursively typed values **roll** w are only manipulated by the base invertible combinators c.

$$w ::= () \mid (w_1, w_2) \mid \mathbf{inl}\ w \mid \mathbf{inr}\ w \mid \mathbf{roll}\ w$$

The semantics of RRARR again takes the form of three relations: one for unidirectional arrows and two for invertible arrows. The first ($\mu\ w_1 \mapsto w_2$) reads that μ maps w_1 to w_2, confirming the intuition that unidirectional arrows represent functions. The second ($\alpha\ w; w_1 \mapsto w_2$) and third ($\alpha\ w; w_1 \leftarrow\!\shortmid w_2$) read that given parameter w, α maps w_1 to w_2 under the forward (resp. backward) evaluation, confirming the intuition that our invertible arrows correspond to parameterized bijections. The rules closely follow the informal descriptions presented in the previous section. We assume a base invertible semantics for combinators c of the form $c\ w_1 \mapsto w_2$, invoked by the rules concerning *arr* for each arrow.

Unidirectional Evaluation $\boxed{\mu\ w_1 \mapsto w_2}$

$$\frac{c\ w_1 \mapsto w_2}{(\mathit{arr}_{\mathrm{u}}\ c)\ w_1 \mapsto w_2} \qquad \frac{\mu_1\ w_1 \mapsto w_2 \quad \mu_2\ w_2 \mapsto w_3}{(\mu_1 \ggg_{\mathrm{u}} \mu_2)\ w_1 \mapsto w_3} \qquad \frac{\mu\ w_1 \mapsto w_2}{(\mathit{first}_{\mathrm{u}}\ \mu)\ (w_1, w_3) \mapsto (w_2, w_3)}$$

$$\frac{\mu\ w_1 \mapsto w_2}{(\mathit{left}_{\mathrm{u}}\ \mu)\ \textbf{inl}\ w_1 \mapsto \textbf{inl}\ w_2} \qquad \frac{}{(\mathit{left}_{\mathrm{u}}\ \mu)\ \textbf{inr}\ w_1 \mapsto \textbf{inr}\ w_1} \qquad \frac{}{\mathit{clone}\ w_1 \mapsto (w_1, w_1)}$$

$$\frac{\alpha\ w; w_1 \mapsto w_2}{(\mathit{run}\ \alpha)\ (w, w_1) \mapsto w_2}$$

Forward (and Backward) Evaluation $\boxed{\alpha\ w; w_1 \mapsto w_2}$ $\left(\boxed{\alpha\ w; w_1 \leftarrow\!\shortmid\ w_2}\right)$

$$\frac{c\ w_1 \mapsto w_2}{(\mathit{arr}_{\mathrm{r}}\ c)\ w; w_1 \mapsto w_2} \qquad \frac{\alpha_1\ w; w_1 \mapsto w_2 \quad \alpha_2\ w; w_2 \mapsto w_3}{(\alpha_1 \ggg_{\mathrm{r}} \alpha_2)\ w; w_1 \mapsto w_3} \qquad \frac{\alpha\ w; w_1 \mapsto w_2}{(\mathit{first}_{\mathrm{r}}\ \alpha)\ w; (w_1, w_3) \mapsto (w_2, w_3)}$$

$$\frac{\alpha\ w; w_1 \mapsto w_2}{(\mathit{left}_{\mathrm{r}}\ \alpha)\ w; \textbf{inl}\ w_1 \mapsto \textbf{inl}\ w_2} \qquad \frac{}{(\mathit{left}_{\mathrm{r}}\ \alpha)\ w; \textbf{inr}\ w_1 \mapsto \textbf{inr}\ w_1} \qquad \frac{\alpha\ w; w_2 \leftarrow\!\shortmid\ w_1}{\alpha^{\dagger}\ w; w_1 \mapsto w_2}$$

$$\frac{\alpha_1\ w; w_1 \mapsto w_2}{(\mathit{case!}\ \alpha_1\ \alpha_2)\ \textbf{inl}\ w; w_1 \mapsto w_2} \qquad \frac{\alpha_2\ w; w_1 \mapsto w_2}{(\mathit{case!}\ \alpha_1\ \alpha_2)\ \textbf{inr}\ w; w_1 \mapsto w_2} \qquad \frac{\mu\ w \mapsto w' \quad \alpha\ w'; w_1 \mapsto w_2}{(\mu \gg! \alpha)\ w; w_1 \mapsto w_2}$$

$$\frac{\alpha\ (w, w_1); w_2 \mapsto w_3}{(\mathit{pin}\ \alpha)\ w; (w_1, w_2) \mapsto (w_1, w_3)}$$

Fig. 4. The semantics of RRARR. As before, the backward evaluation rules are symmetrically obtained from the forward rules.

The semantics satisfies the desired properties of subject reduction and invertibility, although we refer to our mechanized formalization for the details.[9]

4.3 Locally Invertible Interpretation

Recall that our goal is to define a locally invertible interpretation, whereas the straightforward semantics of Section 4.2 depended on a unidirectional evaluation. In this section, we give an alternative interpretation of RRARR, utilizing the reversible reader (RReader) [23] to interpret the invertible arrow combinators.

$$[\![C \cdot A \leftrightsquigarrow B]\!] = \mathsf{RReader}\ C\ A\ B$$

Here, RReader $C\ A\ B$ consists of the bijections of type $C \otimes A \leftrightharpoons C \otimes B$ that keep the C part unchanged. This arrow was originally introduced with the intention of modelling a bijection with some "static" input C [23]. Regarding $\rightsquigarrow$, we use the irreversibility effect [26] that leverages the fact that every unidirectional computation can be simulated by a locally invertible computation yielding "garbage" [8], as:

$$[\![A \rightsquigarrow B]\!] = \exists G.\, A \leftrightharpoons G \otimes B$$

Combining these two effects is a novel point of RRARR; in particular, we contribute the core constructs of *case*!, $\gg$!, *pin* and *run*, which enable communication between the two. Locally invertible interpretations of the primitives in

[9] https://git.sr.ht/~aathn/kalpis-agda

$$\begin{array}{rlrl}
\mathit{unitel}_\times : & 1 \otimes A \leftrightharpoons A & \mathit{assoc}_+ : & A \oplus (B \oplus C) \leftrightharpoons (A \oplus B) \oplus C \\
\mathit{swap}_\times : & A \otimes B \leftrightharpoons B \otimes A & \mathit{distr} : & (A \oplus B) \otimes C \leftrightharpoons A \otimes C \oplus B \otimes C \\
\mathit{assocl}_\times : & A \otimes (B \otimes C) \leftrightharpoons (A \otimes B) \otimes C & \mathit{inl} : & A \leftrightharpoons A \oplus B \\
\mathit{swap}_+ : & A \oplus B \leftrightharpoons B \oplus A & \mathit{roll} : & A[\mu X.A/X] \leftrightharpoons \mu X.A
\end{array}$$

$$\frac{}{\mathit{id} : A \leftrightharpoons A} \qquad \frac{c : A \leftrightharpoons B}{c^\dagger : B \leftrightharpoons A} \qquad \frac{c_1 : A \leftrightharpoons B \qquad c_2 : B \leftrightharpoons C}{c_1 \fatsemi c_2 : A \leftrightharpoons C}$$

$$\frac{c_1 : A \leftrightharpoons C \qquad c_2 : B \leftrightharpoons D}{c_1 \otimes c_2 : A \otimes B \leftrightharpoons C \otimes D} \qquad \frac{c_1 : A \leftrightharpoons C \qquad c_2 : B \leftrightharpoons D}{c_1 \oplus c_2 : A \oplus B \leftrightharpoons C \oplus D}$$

Fig. 5. The invertible primitives of Π^o [26]. Note that we replace the looping construct *trace* with the derived *inl* for simplicity (Section 4.5 recovers the expressiveness of this combinator).

each system have been given in the existing results. Here, we extend the results with the operations novel to rrArr, to show that the two systems together give a locally invertible model of partially invertible computations.

As our target invertible language, we use Π^o [26], whose combinators c constitute a minimal set of (non-total) invertible operations. The combinators support sequential composition ($c_1 \fatsemi c_2$), parallel composition ($c_1 \otimes c_2$ and $c_1 \oplus c_2$), and importantly, a local inversion operator ($c^\dagger$) such that $(c_1 \fatsemi c_2)^\dagger = c_2^\dagger \fatsemi c_1^\dagger$. Figure 5 shows a summary of the primitives; their behavior should be obvious from the types (see the Agda formalization for details).

We now proceed to give another interpretation of the core constructs of rrArr.

Partially invertible branching. Given α_1 and α_2 with $[\![\alpha_1]\!] : C \otimes A \leftrightharpoons C \otimes B$ and $[\![\alpha_2]\!] : D \otimes A \leftrightharpoons D \otimes B$, we must construct

$$[\![\mathit{case!}\ \alpha_1\ \alpha_2]\!] : (C \oplus D) \otimes A \leftrightharpoons (C \oplus D) \otimes B.$$

Using *distr*, we can convert $(C \oplus D) \otimes A$ to $C \otimes A \oplus D \otimes A$, after which $[\![\alpha_1]\!]$ and $[\![\alpha_2]\!]$ can be run in parallel. Factoring out the B, we get the required transformation.

$$[\![\mathit{case!}\ \alpha_1\ \alpha_2]\!] = \mathit{distr} \fatsemi [\![\alpha_1]\!] \oplus [\![\alpha_2]\!] \fatsemi \mathit{distr}^\dagger$$

Pinning. Given α with $[\![\alpha]\!] : (C \otimes D) \otimes A \leftrightharpoons (C \otimes D) \otimes B$, we must produce

$$[\![\mathit{pin}\ \alpha]\!] : C \otimes (D \otimes A) \leftrightharpoons C \otimes (D \otimes B).$$

As the reversible reader arrow $[\![\alpha]\!]$ already returns the context C unchanged, we only need to shuffle the inputs and outputs appropriately.

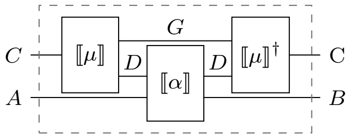
$$[\![\mathit{pin}\ \alpha]\!] = \mathit{assocl}_\times \fatsemi [\![\alpha]\!] \fatsemi \mathit{assocl}_\times^\dagger$$

Partially invertible composition. Given μ and α with $[\![\mu]\!] : C \leftrightharpoons G \otimes D$ and $[\![\alpha]\!] : D \otimes A \leftrightharpoons D \otimes B$, we must construct

$$[\![\mu \ggg!\ \alpha]\!] : C \otimes A \leftrightharpoons C \otimes B.$$

The basic idea is to run $[\![\mu]\!]$ to produce a D-typed value to run $[\![\alpha]\!]$ on, however, this brings with it unwanted garbage. Fortunately, since $[\![\alpha]\!]$ is a reversible reader arrow, it is guaranteed to preserve the D-component, meaning that after running it we have the same D and G-values available to us as before. These can be turned back into the original C value by running $[\![\mu]\!]$ backwards, giving the transformation required.

$$[\![\mu \gg\!! \alpha]\!] = \\ [\![\mu]\!] \otimes id \fatsemi assocl_\times^\dagger \fatsemi id \otimes [\![\alpha]\!] \fatsemi assocl_\times \fatsemi [\![\mu]\!]^\dagger \otimes id$$



Note that this is precisely the construction underlying the reversible updates [5] of imperative reversible languages, and that $[\![\alpha]\!]$ preserving the context is crucial for the construction to succeed.

Running invertible computations. Given α with $[\![\alpha]\!] : C \otimes A \leftrightharpoons C \otimes B$, we must produce

$$[\![run\ \alpha]\!] : C \otimes A \leftrightharpoons G \otimes B,$$

for some G. Clearly it suffices to take $[\![\alpha]\!]$ with $G = C$, and we are done.

4.4 Correctness

We now state the desired correctness properties of our locally invertible interpretation, which show that it is equivalent to the direct semantics of Figure 4 and that $[\![\alpha]\!]$ is indeed a reversible reader arrow (*i.e.*, it preserves the context C).

Theorem 3 (RRARR $\dashrightarrow$ Π^o Soundness).

- $\mu\ w_1 \mapsto w_2$ *implies* $[\![\mu]\!]\ w_1 \mapsto (g, w_2)$ *for some* g.
- $\alpha\ w; w_1 \mapsto w_2$ *implies* $[\![\alpha]\!]\ (w, w_1) \mapsto (w, w_2)$. □

Theorem 4 (RRARR $\dashrightarrow$ Π^o Completeness).

- $[\![\mu]\!]\ w_1 \mapsto (g, w_2)$ *implies* $\mu\ w_1 \mapsto w_2$.
- $[\![\alpha]\!]\ (w, w_1) \mapsto (w', w_2)$ *implies* $w = w'$ *and* $\alpha\ w; w_1 \mapsto w_2$. □

The theorems do not refer to the backward evaluation directly, utilizing the invertibility of both RRARR and Π^o.

4.5 Higher-order Computation

The previous sections laid out the fundamental ideas for representing partial invertibility in a locally invertible setting. However, with RRARR being first-order, it is not sufficient to be able to interpret KALPIS in a simple way. In this section, we extend the language with four new combinators enabling proper higher-order computation, shown in Figure 6. The combinators *curry* and *app* are the standard currying and evaluation maps, creating and applying functions

$$A, B ::= \cdots \mid A \to B \mid A \leftrightarrow B$$

$$\mu ::= \cdots \mid \mathit{curry}\ \mu \mid \mathit{app} \mid \mathit{curry}^\bullet\ \alpha \qquad \alpha ::= \cdots \mid \mathit{app}^\bullet \qquad w ::= \cdots \mid \langle \mu, w \rangle \mid \langle \alpha, w \rangle$$

$$\frac{\mu : C \otimes A \rightsquigarrow B}{\mathit{curry}\ \mu : C \rightsquigarrow (A \to B)} \qquad \frac{\alpha : C \cdot A \leftrightsquigarrow B}{\mathit{curry}^\bullet\ \alpha : C \rightsquigarrow (A \leftrightarrow B)} \qquad \frac{}{\mathit{app} : (A \to B) \otimes A \rightsquigarrow B} \qquad \frac{}{\mathit{app}^\bullet : (A \leftrightarrow B) \cdot A \leftrightsquigarrow B}$$

$$\frac{}{(\mathit{curry}\ \mu)\ w \mapsto \langle \mu, w \rangle} \qquad \frac{}{(\mathit{curry}^\bullet\ \alpha)\ w \mapsto \langle \alpha, w \rangle} \qquad \frac{\mu\ (w, w_1) \mapsto w_2}{\mathit{app}\ (\langle \mu, w \rangle, w_1) \mapsto w_2} \qquad \frac{\alpha\ w; w_1 \mapsto w_2}{\mathit{app}^\bullet\ \langle \alpha, w \rangle; w_1 \mapsto w_2}$$

Fig. 6. Combinators for higher-order computation in RRARR.

$A \to B$. Their invertible counterparts $\mathit{curry}^\bullet$ and $\mathit{app}^\bullet$ provide the final core construct from Section 2: abstraction and application of invertible computations. They operate over parameterized bijections, abstracting the parameter to get a bijection value $A \leftrightarrow B$. The values are extended accordingly with two new closure forms $\langle \mu, w \rangle : A \to B$ and $\langle \alpha, w \rangle : A \leftrightarrow B$, where $\mu : C \otimes A \rightsquigarrow B$, $\alpha : C \cdot A \leftrightsquigarrow B$, and $w : C$, representing staged unidirectional and invertible computations, respectively.

Having higher-order computation in the invertible setting has been challenging [2,12,39,40]. Borrowing the idea from [39,40], we address the issue by leveraging the fact that the function and bijection values are only part of invertible computations as parameters of parameterized bijections; hence, we only need a limited form of higher-orderness. We extend Π^o with two additional primitive operations:

$$\begin{aligned} \mathit{curry}_{\leftrightharpoons} &: (C \otimes A \leftrightharpoons C \otimes B) \to (C \leftrightharpoons C \otimes (A \leftrightarrow B)) \\ \mathit{app}_{\leftrightharpoons} &: ((A \leftrightarrow B) \otimes A) \leftrightharpoons ((A \leftrightarrow B) \otimes B) \end{aligned}$$

The former takes a combinator with an auxiliary piece of "state" C, and abstracts it into a bijection given a value of C. The latter applies a bijection, and saves it to enable reversing the operation later. To represent the values of type $A \leftrightarrow B$ in Π^o, we introduce a third form of closure $\langle f, w \rangle$, where we have $f : C \otimes A \leftrightharpoons C \otimes B$ and $w : C$. Then, the semantics of $\mathit{app}_{\leftrightharpoons}$ and $\mathit{curry}_{\leftrightharpoons}$ are as follows:

$$\frac{\mathit{clos} = \langle f, w \rangle}{(\mathit{curry}_{\leftrightharpoons}\ f)\ w \mapsto (w, \mathit{clos})} \qquad \frac{f\ (w, a) \mapsto (w', b)}{\mathit{app}_{\leftrightharpoons}\ (\langle f, w \rangle, a) \mapsto (\langle f, w' \rangle, b)}$$

As before, the inverse semantics is symmetric; *e.g.,* $(\mathit{curry}_{\leftrightharpoons}\ f)^\dagger\ (w, \mathit{clos}) \mapsto w$ if $\mathit{clos} = \langle f, w \rangle$. The (non-total) invertibility of $\mathit{curry}_{\leftrightharpoons}$ is trivial, as its inverse fails unless its input matches the corresponding output; it is essentially a unidirectional function embedded in the invertible world. Since observational equality of closure values is undecidable, the equality check must rely on some other, intensional (*e.g.,* syntactic) equality. Practically, this means that the combinator can only be used to create a closure and then subsequently undo the very same closure. However, this does not pose an issue for the translation from RRARR, where closures will only result from uses of *curry* and *curry*$^\bullet$, both of which are unidirectional arrows ($\rightsquigarrow$). These unidirectional arrows will only be executed backwards as part of partially invertible compositions ($\gg !$), which ensures that the input is the same as the corresponding output.

Now, we can interpret $[\![app]\!] = app_{\leftrightharpoons}$, $[\![app^\bullet]\!] = app_{\leftrightharpoons}$, and

$$[\![curry\ \mu]\!] = inl \mathbin{⨟} curry_{\leftrightharpoons}\ (inl^\dagger \otimes id \mathbin{⨟} [\![\mu]\!] \mathbin{⨟} inr \otimes id), \quad [\![curry^\bullet\ \alpha]\!] = curry_{\leftrightharpoons}\ [\![\alpha]\!].$$

The former construction curries $[\![\mu]\!] : C \otimes A \leftrightharpoons G \otimes B$ given $w : C$ by creating a one-shot closure $\langle f, \mathbf{inl}\ w\rangle$ which turns into $\langle f, \mathbf{inr}\ g\rangle$ for $g : G$ when first applied, and fails on a second application.

The theorems of Section 4.4 extend without difficulty to the higher-order combinators, although the statement is somewhat more intricate due to the differing set of closure values between RRARR and Π^o. We refer to the mechanized formalization in Agda for details.

5 Interpreting KALPIS with Arrows

Theorem 1 (Section 3.6) suggests that a unidirectional term-in-context $\Gamma \vdash u : A$ can be seen as a function from Γ to A, and that an invertible term-in-context $\Gamma; \Theta \vdash r : A$ can be seen as a bijection between Θ and A parameterized by Γ. Then, it is natural that they be related with the two arrows $(- \rightsquigarrow -)$ and $(- \cdot - \leftrightsquigarrow -)$ of RRARR, respectively. In this section, we give a formal account of this relation by translating terms of KALPIS into RRARR, giving by extension a compositional locally invertible interpretation of KALPIS.

We first define some operations on typing contexts. We define $\Gamma^\times$ as

$$(x_1 : A_1, \ldots, x_n : A_n)^\times = (((1 \otimes A_1) \otimes A_2) \otimes \cdots) \otimes A_n.$$

It is straightforward to define an operator $lookup_x : \Gamma^\times \rightsquigarrow A$ provided that $\Gamma(x) = A$. We also use a combinator $split_{\Theta_1,\Theta_2} : (\Theta_1 \uplus \Theta_2)^\times \leftrightharpoons \Theta_1^\times \otimes \Theta_2^\times$ for splitting the linear environments. Then, we give two type-directed transformations: $\Gamma \vdash u : A \dashrightarrow \mu$ that transforms u to μ of type $\Gamma^\times \rightsquigarrow A$, and $\Gamma; \Theta \vdash r : A \dashrightarrow \alpha$ that transforms r to α of type $\Gamma^\times \cdot \Theta^\times \leftrightsquigarrow A$. For the purposes of the translation, we consider a fixed set of type constructors $\mathsf{T}\ \overline{B} ::= 1 \mid A \otimes B \mid A \oplus B \mid \mathsf{Rec}_A$, identifying $\mu X.A$ with Rec_A.

Without loss of generality, we drop unnecessary **with**-conditions, so that a **case**$^\bullet$-expression with one branch needs no **with**-clause, and one with two branches needs only one clause. Due to the space limitations, we present only the most representative cases here, and point the interested reader to the mechanized formalization in Agda.[10]

[10] https://git.sr.ht/~aathn/kalpis-agda

Case T-UCase ($A \oplus B$).

$$\frac{\begin{array}{c}\Gamma \vdash u : A \oplus B \dashrightarrow \mu \\ \Gamma, x : A; \Theta \vdash r_1 : C \dashrightarrow \alpha_1 \qquad \Gamma, y : B; \Theta \vdash r_2 : C \dashrightarrow \alpha_2\end{array}}{\begin{array}{l}\Gamma; \Theta \vdash \textbf{case}\ u\ \textbf{of}\ \mathsf{InL}\ x \to r_1;\ \mathsf{InR}\ y \to r_2 : C \dashrightarrow \\ \qquad (\mathit{clone} \ggg_{\mathrm{u}} \mathit{first}_{\mathrm{u}}\ \mu \ggg_{\mathrm{u}} \mathit{arr}_{\mathrm{u}}\ (\mathit{swap}_{\times} \fatsemi \mathit{distl})) \gg! \mathit{case}!\ \alpha_1\ \alpha_2\end{array}}$$

We can duplicate $\Gamma^{\times}$ using *clone* and use one copy to construct $A \oplus B$ with μ. Using $\mathit{distl} : A \otimes (B \oplus C) \leftrightharpoons A \otimes B \oplus A \otimes C$, which is easily derived, we distribute the second copy of Γ over the sum. Then, the required combinator can be constructed through a combination of partially invertible composition ($\gg!$) and branching (*case*!), where we have $\mathit{case}!\ \alpha_1\ \alpha_2 : (\Gamma^{\times} \otimes A \oplus \Gamma^{\times} \otimes B) \cdot \Theta \leftrightsquigarrow C$.

Case T-RCase ($A \oplus B$).

$$\frac{\begin{array}{c}\Gamma; \Theta_1 \vdash r_1 : A \oplus B \dashrightarrow \alpha_1 \qquad \Gamma; \Theta_2, x : A \vdash r_2 : C \dashrightarrow \alpha_2 \\ \Gamma; \Theta_2, y : B \vdash r_3 : C \dashrightarrow \alpha_3 \qquad \Gamma \vdash u : C \to \mathsf{Bool} \dashrightarrow \mu\end{array}}{\begin{array}{l}\Gamma; \Theta_1 \uplus \Theta_2 \vdash \textbf{case}^{\bullet}\ r_1\ \textbf{of}\ \mathsf{InL}\ x \to r_2;\ \mathsf{InR}\ y \to r_3\ \textbf{with}\ u : C \dashrightarrow \\ \qquad \mathit{arr}_{\mathrm{r}}\ \mathit{split}_{\Theta_1,\Theta_2} \ggg_{\mathrm{r}} \mathit{first}_{\mathrm{r}}\ \alpha_1 \ggg_{\mathrm{r}} \mathit{arr}_{\mathrm{r}}\ (\mathit{swap}_{\times} \fatsemi \mathit{distl}) \ggg_{\mathrm{r}} \\ \qquad \mathit{case}\ \alpha_2\ \alpha_3\ (\mathit{mkCond}\ \mu)\end{array}}$$

The idea is similar to T-UCase, but we now operate in the invertible world, so we split $(\Theta_1 \uplus \Theta_2)^{\times}$ instead of duplicating Γ, and compose using $\ggg_{\mathrm{r}}$ instead of $\gg!$. The combinator $\mathit{case}\ \alpha_1\ \alpha_2\ \alpha_3 \triangleq \mathit{left}_{\mathrm{r}}\ \alpha_1 \ggg_{\mathrm{r}} \mathit{right}_{\mathrm{r}}\ \alpha_2 \ggg_{\mathrm{r}} \alpha_3^{\dagger}$ with type

$$\mathit{case} : (D \cdot A \leftrightsquigarrow C) \to (D \cdot B \leftrightsquigarrow C) \to (D \cdot C \leftrightsquigarrow C \oplus C) \to D \cdot (A \oplus B) \leftrightsquigarrow C,$$

provides an invertible branching operator analogous to *case*!, with a postcondition for merging the branches. We convert $\mu : \Gamma^{\times} \rightsquigarrow (C \to \mathsf{Bool})$ to an arrow $\mathit{mkCond}\ \mu : \Gamma^{\times} \cdot C \leftrightsquigarrow C \oplus C$ through the *mkCond* operator, which can be defined using *pin*, *case*! and *app* in tandem.

Cases T-Abs$^{\bullet}$, T-RApp.

$$\frac{\Gamma; x : A \vdash r : B \dashrightarrow \alpha}{\begin{array}{l}\Gamma \vdash \lambda^{\bullet} x.r : A \leftrightarrow B \dashrightarrow \\ \qquad \mathit{curry}^{\bullet}\ (\mathit{arr}_{\mathrm{r}}\ \mathit{unitel}_{\times}^{\dagger} \ggg_{\mathrm{r}} \alpha)\end{array}} \qquad \frac{\Gamma \vdash u : A \leftrightarrow B \dashrightarrow \mu \qquad \Gamma; \Theta \vdash r : A \dashrightarrow \alpha}{\Gamma; \Theta \vdash u \diamond r : B \dashrightarrow \alpha \ggg_{\mathrm{r}} (\mu \gg! \mathit{app}^{\bullet})}$$

For T-Abs$^{\bullet}$, we get $\alpha : \Gamma^{\times} \cdot 1 \otimes A \leftrightsquigarrow B$, which we $\mathit{curry}^{\bullet}$ after handling the unit. For T-RApp, α transforms $\Theta^{\times}$ to A, letting μ be applied through a partially invertible composition ($\gg!$) with $\mathit{app}^{\bullet}$.

Case T-Pin.

$$\frac{\Gamma \vdash u : C \to A \leftrightarrow B \dashrightarrow \mu \qquad \Gamma; \Theta \vdash r : C \otimes A \dashrightarrow \alpha}{\Gamma; \Theta \vdash \mathit{pin}\ u \diamond r : C \otimes B \dashrightarrow \alpha \ggg_{\mathrm{r}} \mathit{pin}\ ((\mathit{first}_{\mathrm{u}}\ \mu \ggg_{\mathrm{u}} \mathit{app}) \gg! \mathit{app}^{\bullet})}$$

We have α producing $C \otimes A$, and with parameter $\Gamma^{\times} \otimes C$, we can apply μ to produce B. Thus, we must shift C from the output into the parameter, and *pin* achieves just that.

Correctness. Finally, we show the correctness of the translation with respect to the semantics of Sections 3.5 and 4.2. Before we state correctness, we must first define a translation of the values, since they differ between KALPIS and RRARR.

$$\llbracket () \rrbracket = (), \quad \llbracket (v_1, v_2) \rrbracket = (\llbracket v_1 \rrbracket, \llbracket v_2 \rrbracket),$$
$$\llbracket \mathsf{InL}\ v \rrbracket = \mathbf{inl}\ \llbracket v \rrbracket, \quad \llbracket \mathsf{InR}\ v \rrbracket = \mathbf{inr}\ \llbracket v \rrbracket, \quad \llbracket \mathsf{Roll}\ v \rrbracket = \mathbf{roll}\ \llbracket v \rrbracket,$$
$$\llbracket \langle \lambda x.u, \gamma \rangle \rrbracket = \langle \llbracket u \rrbracket, \llbracket \gamma \rrbracket \rangle, \quad \llbracket \langle \lambda^{\bullet} x.r, \gamma \rangle \rrbracket = \langle arr\ unitel_{\times}^{\dagger} \ggg_{\mathrm{r}} \llbracket r \rrbracket, \llbracket \gamma \rrbracket \rangle$$

The base values are translated trivially, whereas the closures are translated according to the type-directed translation given above (cf. **Case** T-ABS$^{\bullet}$). We also define a translation of value environments γ in the obvious way.

Then, we can state the correctness of the translation as below.

Theorem 5 (KALPIS ⇢ RRARR Soundness).
- $\Gamma \vdash u : A \dashrightarrow \mu$ *and* $\gamma \vdash u \Downarrow v$ *implies* $\mu\ \llbracket \gamma \rrbracket \mapsto \llbracket v \rrbracket$
- $\Gamma; \Theta \vdash r : A \dashrightarrow \alpha$ *and* $\gamma; \theta \vdash r \Rightarrow v$ *implies* $\alpha\ \llbracket \gamma \rrbracket; \llbracket \theta \rrbracket \mapsto \llbracket v \rrbracket$. □

This theorem does not refer to the backward evaluation directly, utilizing the invertibility of both KALPIS and RRARR. The completeness part, on the other hand, does need a separate statement for the backward direction, since there is no *a priori* guarantee that the output w is of the form $\llbracket \theta \rrbracket$.

Theorem 6 (KALPIS ⇢ RRARR Completeness).
- $\Gamma \vdash u : A \dashrightarrow \mu$ *and* $\mu\ \llbracket \gamma \rrbracket \mapsto w$ *implies* $\gamma \vdash u \Downarrow v$ *for* v *with* $\llbracket v \rrbracket = w$.
- $\Gamma; \Theta \vdash r : A \dashrightarrow \alpha$ *and* $\alpha\ \llbracket \gamma \rrbracket; \llbracket \theta \rrbracket \mapsto w$ *implies* $\gamma; \theta \vdash r \Rightarrow v$ *for* v *with* $\llbracket v \rrbracket = w$.
- $\Gamma; \Theta \vdash r : A \dashrightarrow \alpha$ *and* $\alpha\ \llbracket \gamma \rrbracket; \llbracket v \rrbracket \leftarrow\!\shortmid w$ *implies* $\gamma; v \vdash r \Leftarrow \theta$ *for* θ *with* $\llbracket \theta \rrbracket = w$. □

We refer to the Agda code in the supplementary material for the proofs.

6 Related Work

KALPIS and RRARR are not the first to support partial invertibility. In the imperative setting, languages such as Janus [35, 53], Frank's R [17], and R-While [19] support a limited form of partial invertibility via reversible update operators [6]. An example of a reversible update statement is x += e, whose effect can be reverted by the corresponding inverse statement x -= e. Both statements use the same e, which need not be invertible (*e.g.*, x += yz is reverted by x -= yz, and vice versa). In the functional setting, Theseus [27] allows a bijection to take additional parameters, but only provided that they are available at compile time. RFun version 2,[11] an extension to the original RFun [54], and CoreFun [25] allow more flexibility via so-called ancilla parameters, which are translated to auxiliary inputs and outputs of the invertible computation. Their approach is similar to KALPIS's but more restrictive since they lack support for the *pin* operator and

[11] https://github.com/kirkedal/rfun-interp

higher-order computation. Jeopardy [31] is a recent invertible language where even irreversible functions can be inverted in certain contexts depending on implicitly available information. However, this is still work in progress, and seems to lean closer to program inversion methods than the lightweight type-based approach we employ.

Sparcl [39,40] is the most flexible system that supports partial invertibility to our knowledge, which is realized through a more advanced language foundation. Instead of bijections $A \leftrightarrow B$, Sparcl features *invertible data* marked by the type $A^\bullet$, which implicitly corresponds to some bijection $S \leftrightarrow A$. This idea of invertible data is inherited from the HOBiT language [38], which represents lens combinators [15, 16] as higher-order functions to achieve applicative-style higher-order bidirectional programming [36, 37]. The type system of Sparcl ensures that a closed linear function between invertible data $!(A^\bullet \multimap B^\bullet)$ is isomorphic to a (non-total) bijection between A and B, so that partial invertibility can be represented as a function that takes both unidirectional and invertible data $C \to A^\bullet \multimap B^\bullet$. This representation affords more flexibility than Kalpis does: invertible data is allowed to be captured in abstractions, and can even appear in subcomponents of datatypes (*e.g.*, $\mathsf{Int} \otimes (\mathsf{Int}^\bullet)$ or $\mathsf{Int} \oplus (\mathsf{Int}^\bullet)$ are both valid types). However, this flexibility comes at the cost of complexity, requiring a semantics that interleaves partial evaluation and invertible computation, making a locally invertible interpretation difficult. We remark that the holed residuals $\langle x.E \rangle$ featured in Sparcl's core system bear a strong resemblance to bijections $\lambda^\bullet x.r$ in Kalpis.

Our combinator language rrArr can be seen as an extension of ML_Π, an arrow metalanguage on top of the invertible language Π treating information creation and loss (non-totality and irreversibility) as an effect [26]. By combining their work with the reversible reader arrow [23], we are able to give erasing (weakening) as a derived operation defined via the operator *run* (as demonstrated in Section 4). Further research on the nontrivial interaction between the arrows, such as an equational characterization and a denotational model, is left for future work. While the previous work is able to treat non-totality as part of an effect, we assume some non-total operations in the underlying invertible system due to the inclusion of recursive and functional types.

The design of Kalpis is inspired by the arrow calculus of Lindley, Wadler, and Yallop [33], which is a metalanguage for the conventional representation of arrows [24], analogous to the monad metalanguage [42]. In a sense, Kalpis can be seen as a counterpart to the arrow calculus for rrArr. For example, the treatment of $\lambda^\bullet x.r$ is actually inherited from the arrow calculus, where arrows cannot be nested in general [34], unless the underlying arrow supports application to form a monad [24]. To the best of our knowledge, a monad-based programming system for invertible/reversible computation does not exist, though there are some closely related results, including monads for nondeterministic computation (such as [14]) and a monadic programming framework for bidirectional transformations [20, 52]. However, these existing approaches lack the guarantee of bijectivity—a motivation to use invertible languages.

The importance of partial invertibility has been recognized in the neighboring literature on program inversion—program transformations that derive a program of f^{-1} for a given program of f. Partial inversion [44, 47] essentially applies a binding-time analysis [21, 28] to an input program, where the static data can be treated as unidirectional inputs. The technique is further extended to treat results of inverses as unidirectional [3, 29, 30]. This treatment is similar to the role of *pin* in Kalpis and Sparcl [39,40] in that it converts invertible data into "static" parameters. Some approaches to program inversion are more liberal: semi inversion [41] essentially converts a program into a logic program, where there is no clear boundary between unidirectional and invertible data, and the PINS system [49], in addition to an original program, can take a control structure of an inverse program to effectively synthesize inverses that may not mirror the control structures of the original. The main limitation of program inversion is that as a program transformation it may fail, often for reasons that are not obvious to programmers.

7 Conclusion

We have presented a set of four core constructs for partially invertible programming, demonstrated their expressiveness through examples, and shown that they can be given a locally invertible interpretation, thus solving an open problem in the field. The four constructs are (1) partially invertible branching, (2) pinning invertible inputs, (3) partially invertible composition, and (4) abstraction and application of invertible computations. We designed the partially invertible language Kalpis on top of these constructs and formalized its syntax, type system and operational semantics. We then presented rrArr, a low-level arrow language with primitives directly corresponding to the constructs, and gave it a locally invertible interpretation based on two effects—the irreversibility effect [26] and the reversible reader [23]. Finally, we presented a type-directed translation from Kalpis to rrArr, showing how to support expressive partial invertibility on top of a locally invertible foundation. Proofs of all theorems stated in the paper are formalized by the accompanying Agda code.[12]

Acknowledgments. We thank Eijiro Sumii, Oleg Kiselyov, and other Sumii-Matsuda Lab members for useful feedback on a preliminary version of this research. This work is partially supported by JSPS KAKENHI Grant Numbers, JP19K11892, JP20H04161, and JP22H03562, EPSRC Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1), and Royal Society Grant *Bidirectional Compiler for Software Evolution* (IES\R3\170104). This work was also partially supported by a scholarship awarded to the first author by the Marianne and Marcus Wallenberg Foundation (SJF application BA21-0019).

Data Availability. The accompanying artifact that contains the prototype implementation of Kalpis and the Agda formalization mentioned in this paper is available from https://doi.org/10.5281/zenodo.10511566.

[12] https://git.sr.ht/~aathn/kalpis-agda

References

1. Abel, A., Chapman, J.: Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In: Levy, P., Krishnaswami, N. (eds.) Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. EPTCS, vol. 153, pp. 51–67 (2014). https://doi.org/10.4204/EPTCS.153.4
2. Abramsky, S.: A structural approach to reversible computation. Theor. Comput. Sci. **347**(3), 441–464 (2005). https://doi.org/10.1016/j.tcs.2005.07.002
3. Almendros-Jiménez, J.M., Vidal, G.: Automatic partial inversion of inductively sequential functions. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4449, pp. 253–270. Springer (2006). https://doi.org/10.1007/978-3-540-74130-5_15
4. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM **47**(4), 776–822 (2000). https://doi.org/10.1145/347476.347484
5. Axelsen, H.B., Glück, R.: What do reversible programs compute? In: Hofmann, M. (ed.) Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6604, pp. 42–56. Springer (2011). https://doi.org/10.1007/978-3-642-19805-2_4
6. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4649, pp. 56–69. Springer (2007). https://doi.org/10.1007/978-3-540-74510-5_9
7. Baker, H.G.: NREVERSAL of fortune - the thermodynamics of garbage collection. In: Bekkers, Y., Cohen, J. (eds.) Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings. Lecture Notes in Computer Science, vol. 637, pp. 507–524. Springer (1992). https://doi.org/10.1007/BFb0017210
8. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development **17**(6), 525–532 (11 1973). https://doi.org/10.1147/rd.176.0525
9. Bernardy, J., Boespflug, M., Newton, R.R., Peyton Jones, S., Spiwack, A.: Linear haskell: practical linearity in a higher-order polymorphic language. PACMPL **2**(POPL), 5:1–5:29 (2018). https://doi.org/10.1145/3158093
10. Bowman, W.J., James, R.P., Sabry, A.: Dagger traced symmetric monoidal categories and reversible programming. Work-in-progress report in the 3rd Workshop on Reversible Computation (July 2011), available from https://www.williamjbowman.com/resources/cat-rev.pdf (visited Jan 23, 2024)
11. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science **1**(2), Article No. 1 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005
12. Chen, C., Sabry, A.: A computational interpretation of compact closed categories: reversible programming with negative and fractional types. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434290

13. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001). https://doi.org/10.1145/382780.382785
14. Fischer, S., Kiselyov, O., Shan, C.: Purely functional lazy nondeterministic programming. J. Funct. Program. **21**(4-5), 413–465 (2011). https://doi.org/10.1017/S0956796811000189
15. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005. pp. 233–246. ACM (2005). https://doi.org/10.1145/1040305.1040325
16. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), Article No. 17 (2007). https://doi.org/10.1145/1232420.1232424
17. Frank, M.P.: The R programming language and compiler. MIT Reversible Computing Project Memo #M8, MIT AI Lab (7 1997), available on: https://github.com/mikepfrank/Rlang-compiler/blob/master/docs/MIT-RCP-MemoM8-RProgLang.pdf
18. Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Ohori, A. (ed.) Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2895, pp. 246–264. Springer (2003). https://doi.org/10.1007/978-3-540-40018-9_17
19. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Computer Software **33**(3), 3_108–3_128 (2016). https://doi.org/10.11309/jssst.33.3_108
20. Goldstein, H., Frohlich, S., Wang, M., Pierce, B.C.: Reflecting on random generation. Proc. ACM Program. Lang. **7**(ICFP) (aug 2023). https://doi.org/10.1145/3607842
21. Gomard, C.K., Jones, N.D.: A partial evaluator for the untyped lambda-calculus. J. Funct. Program. **1**(1), 21–69 (1991). https://doi.org/10.1017/S0956796800000058
22. Heunen, C., Kaarsgaard, R.: Quantum information effects. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022). https://doi.org/10.1145/3498663
23. Heunen, C., Kaarsgaard, R., Karvonen, M.: Reversible effects as inverse arrows. Electronic Notes in Theoretical Computer Science **341**, 179–199 (2018). https://doi.org/10.1016/j.entcs.2018.11.009
24. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
25. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun : A typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11106, pp. 304–321. Springer (2018). https://doi.org/10.1007/978-3-319-99498-7_21
26. James, R.P., Sabry, A.: Information effects. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 73–84. ACM (2012). https://doi.org/10.1145/2103656.2103667

27. James, R.P., Sabry, A.: Theseus: a high level language for reversible computing. Work-in-progress report in the 6th Conference on Reversible Computation (2014), available from https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf (visited Jan 23, 2024)
28. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall international series in computer science, Prentice Hall (1993)
29. Kirkeby, M.H., Glück, R.: Inversion framework: Reasoning about inversion by conditional term rewriting systems. In: PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020. pp. 9:1–9:14. ACM (2020). https://doi.org/10.1145/3414080.3414089
30. Kirkeby, M.H., Glück, R.: Semi-inversion of conditional constructor term rewriting systems. In: Gabbrielli, M. (ed.) Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12042, pp. 243–259. Springer (2019). https://doi.org/10.1007/978-3-030-45260-5_15
31. Kristensen, J.T., Kaarsgaard, R., Thomsen, M.K.: Jeopardy: An invertible functional programming language. Work-in-progress paper presented at 34th Symposium on Implementation and Application of Functional Languages **abs/2209.02422** (2022). https://doi.org/10.48550/arXiv.2209.02422
32. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 183–191 (1961). https://doi.org/10.1147/rd.53.0183
33. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. J. Funct. Program. **20**(1), 51–69 (2010). https://doi.org/10.1017/S095679680999027X
34. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electron. Notes Theor. Comput. Sci. **229**(5), 97–117 (2011). https://doi.org/10.1016/j.entcs.2011.02.018
35. Lutz, C.: Janus: a time-reversible language (1986), Letter to R. Landauer. Available on: http://tetsuo.jp/ref/janus.pdf
36. Matsuda, K., Wang, M.: Applicative bidirectional programming with lenses. In: Fisher, K., Reppy, J.H. (eds.) ICFP. pp. 62–74. ACM (2015). https://doi.org/10.1145/2784731.2784750
37. Matsuda, K., Wang, M.: Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. J. Funct. Program. **28**, e15 (2018). https://doi.org/10.1017/S0956796818000096
38. Matsuda, K., Wang, M.: Hobit: Programming lenses without using lens combinators. In: Ahmed, A. (ed.) ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 31–59. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_2
39. Matsuda, K., Wang, M.: Sparcl: A language for partially-invertible computation. Proc. ACM Program. Lang. **4**(ICFP) (8 2020). https://doi.org/10.1145/3409000
40. Matsuda, K., Wang, M.: Sparcl: A language for partially-invertible computation. J. Funct. Program. (in press). https://doi.org/10.1017/S0956796823000126
41. Mogensen, T.Æ.: Semi-inversion of guarded equations. In: Glück, R., Lowry, M.R. (eds.) Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3676, pp. 189–204. Springer (2005). https://doi.org/10.1007/11561347_14

42. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
43. Mu, S., Hu, Z., Takeichi, M.: An injective language for reversible computation. In: Kozen, D., Shankland, C. (eds.) Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3125, pp. 289–313. Springer (2004). https://doi.org/10.1007/978-3-540-27764-4_16
44. Nishida, N., Sakai, M., Sakabe, T.: Partial inversion of constructor term rewriting systems. In: Giesl, J. (ed.) Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3467, pp. 264–278. Springer (2005). https://doi.org/10.1007/978-3-540-32033-3_20
45. Pierce, B.C.: Types and programming languages. MIT Press (2002)
46. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4), 363–397 (1998). https://doi.org/10.1023/A:1010027404223
47. Romanenko, A.: Inversion and metacomputation. In: Consel, C., Danvy, O. (eds.) Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991. pp. 12–22. ACM (1991). https://doi.org/10.1145/115865.115868
48. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. 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. 348–364. Springer (2018). https://doi.org/10.1007/978-3-319-89366-2_19
49. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 492–503. ACM (2011). https://doi.org/10.1145/1993498.1993557
50. Stinson, D., Paterson, M.: Cryptography: Theory and Practice. Textbooks in Mathematics, CRC Press (2018)
51. Wadler, P.: A taste of linear logic. In: Borzyszkowski, A.M., Sokolowski, S. (eds.) Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings. Lecture Notes in Computer Science, vol. 711, pp. 185–210. Springer (1993). https://doi.org/10.1007/3-540-57182-5_12
52. Xia, L., Orchard, D., Wang, M.: Composing bidirectional programs monadically. 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. 147–175. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_6
53. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008. pp. 43–54. ACM (2008). https://doi.org/10.1145/1366230.1366239

54. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: Vos, A.D., Wille, R. (eds.) RC. Lecture Notes in Computer Science, vol. 7165, pp. 14–29. Springer (2011). https://doi.org/10.1007/978-3-642-29517-1_2

Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping[⋆]

Hiroya Fujinami[1,2](✉) and Ichiro Hasuo[1,2]

[1] National Institute of Informatics, Tokyo, Japan
makenowjust@nii.ac.jp

[2] SOKENDAI (The Graduate University for Advanced Studies), Hayama, Japan

Abstract. *Regular expression (regex) matching* is fundamental in many applications, especially in web services. However, matching by *backtracking*—preferred by most real-world implementations for its practical performance and backward compatibility—can suffer from so-called *catastrophic backtracking*, which makes the number of backtracking super-linear and leads to the well-known ReDoS vulnerability. Inspired by a recent algorithm by Davis et al. that runs in linear time for (non-extended) regexes, we study efficient backtracking matching for regexes with two common extensions, namely *look-around* and *atomic grouping*. We present linear-time backtracking matching algorithms for these extended regexes. Their efficiency relies on *memoization*, much like the one by Davis et al.; we also strive for smaller memoization tables by carefully trimming their range. Our experiments—we used some real-world regexes with the aforementioned extensions—confirm the performance advantage of our algorithms.

Keywords: regular expression · look-around · atomic grouping · pattern matching · ReDoS · memoization

1 Introduction

Regex Matching *Regular expressions* (*regexes*) are a fundamental formalism for various pattern-matching tasks. Many regex matching implementations, however, suffer from occasional super-linear growth of their execution time. Such excessive execution time can be exploited for DoS attacks—this is a vulnerability called *regex denial of service* (*ReDoS*). ReDoS is recognized as a significant security concern in many real-world systems, especially web services such as Stack Overflow and Cloudflare (see §2.4 for more details).

Need for Efficient Backtracking Regex Matching The principal cause of ReDoS is *catastrophic backtracking*, that is, the explosion of recursion in a backtracking-based matching algorithm.

⋆ The authors are supported by CREST ZT-IoT Project (No. JPMJCR21M3), ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), and ASPIRE Grant No. JPMJAP2301, JST.

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 90–118, 2024.
https://doi.org/10.1007/978-3-031-57267-8_4

In regex matching, in general, a regex r is converted into a non-deterministic finite automaton (NFA) $\mathcal{A}$, and the latter is executed for an input string w. The non-determinism of $\mathcal{A}$ can be resolved in either a *depth-first* or a *breadth-first* manner. The former is called *backtracking regex matching*, and the latter is the *on-the-fly DFA construction*.

Catastrophic backtracking and ReDoS are phenomena unique to the former (i.e., backtracking)—as is well-known, the time complexity of the on-the-fly DFA construction is linear (i.e., $O(|w|)$). Indeed, many modern regex implementations are based on the on-the-fly DFA construction, including RE2[3], Go's `regexp`[4], and Rust's `regex`[5].

It is practically essential, however, to make backtracking regex matching more efficient. A principal reason is *consistency*. Most existing regex matching implementations use backtracking, and they return only one matching position out of many (see §2.3). While it is possible to replace them with on-the-fly DFA matching, it is non-trivial to ensure consistency, that is, that the chosen matching position is the same as the original backtracking matching implementation. .NET's regex implementation has a linear-time complexity backend using a derivative-based approach, which is compatible with a backtracking backend. Still, it does not support look-around and atomic grouping [28]. Once the returned matching position changes, it can unexpectedly affect the behavior of all the systems (e.g., web services) that use regex matching.

Another reason for improving backtracking regex matching is its *extensibility*. There are many extensions of regexes widely used—such as the ones we study, namely look-around and atomic grouping—and they are supported by few on-the-fly DFA matching implementations.

Existing Work: Linear-time Backtracking Matching with Memoization *Memoization* is a well-known technique for speeding up recursive computations. The recent work [10] shows that memoization can be applied to backtracking regex matching with consistency in mind. Specifically, the work [10] presents a backtracking matching algorithm that runs in $O(|w|)$ time—thus, it is theoretically guaranteed to avoid catastrophic backtracking—for regexes without extensions. (They also mention application to extended regexes in [10], but we found issues in their discussion—see Remark 2).

Our Contribution: Linear-time Backtracking Matching for Some Extended Regexes In this paper, we present a linear-time backtracking matching algorithm for regexes with *look-around* and *atomic grouping*, two real-world extensions of regexes. It uses memoization in order to achieve a linear-time complexity. We also prove that it is consistent (i.e., it chooses the same matching position as the original algorithm without memoization).

The technical key to our algorithm is the design of suitable memoization tables. We follow the general idea in [10] of using memoization for backtracking matching, but our examination of its issues with extended regexes (Remark 2)

[3] `https://github.com/google/re2`
[4] `https://pkg.go.dev/regexp`
[5] `https://docs.rs/regex/latest/regex/`

shows that the range—i.e., the set of possible entries—of memoization tables should be suitably extended. Specifically, the range in [10] is $\{\mathbf{false}\}$, recording only matching failures; it is extended in our algorithm to $\{\mathsf{Failure}(j) \mid j \in \{0,\dots,\nu(\mathcal{A})\}\} \cup \{\mathsf{Success}\}$. Here, $\nu(\mathcal{A})$ is the maximum nesting depth of atomic grouping for the (extended) NFA $\mathcal{A}$, defined in §5.

Our development is rigorous and systematic, based on the notion of NFA whose labels can themselves be NFAs. This extended notion of NFA is suggested in [10, Section IX.B]; in this paper, we formalize it and build its theory.

We experimentally evaluate our algorithm; the experiment results confirm its performance advantages. Additionally, we survey the usage status of look-around and atomic grouping—two regex extensions of our interest—in real-world regexes and demonstrate their wide usage (§6).

Technical Contributions We summarize our technical contributions.

– We propose a backtracking matching algorithm for regexes with look-around, proving its linear-time complexity (§4). This algorithm fixes the issues in the algorithm in [10] (Remark 2) and restores correctness and linearity.
– We also propose a backtracking matching algorithm for regexes with atomic grouping, proving its linear-time complexity (§4).
– We experimentally confirm the performance of our algorithms (§6).
– We investigate the usage status of look-around and atomic grouping in real-world regexes and confirm their wide usage (§6).
– We establish a rigorous theoretical basis for our algorithms for extended regexes, namely NFAs with sub-automata (§2.6).

Organization We provide some preliminaries in §2, such as regex extensions of our interest. Our formalization of NFAs with sub-automata is also presented here. In §3, we discuss the work [10] that is closest to ours. We present our matching algorithm for regex with look-around in §4 and the one for regex with atomic grouping in §5. Then, we discuss our implementation and experimental evaluation in §6. We conclude in §7.

Some additional proofs and other materials are deferred to the appendices in the extended version [15].

Related Work Many related works are discussed elsewhere in suitable contexts. Here, we discuss other related works.

There are many theoretical studies on look-around and atomic grouping. The work [27] is a theoretical study of look-ahead operators; it shows how to convert them to finite automata. Another conversion based on derivatives is introduced in [26]. The work [3] conducts a fine-grained analysis of the size of DFAs obtained from converting regexes with look-ahead, improving the bounds given in [26,27]. The work [5] discusses the relation between look-ahead operators and back-references in regexes. A recent study [22] presents a linear-time matching algorithm for regexes with look-around; it uses a memoization-like construct for efficiency. However, the compatibility with backtracking is not a concern there, unlike the current work. On atomic grouping, conversion to finite automata is proposed [4], where atomic grouping is simulated by look-ahead.

Another common regex extension is *back-reference*. We do not deal with this extension because 1) this extension is known to be non-regular (i.e., the language class defined by back-reference is beyond regular), and 2) its matching problem is known to be NP-complete [1] (thus the search for linear-time matching is doomed). There are other extensions (absent operators, conditional branching, etc.), but they are used less often (cf. §6).

ReDoS countermeasures are an active scientific topic. Besides efficient matching, there are two directions for them: *ReDoS detection* and *ReDoS repair*. ReDoS detection is a problem that determines whether a given regex can cause catastrophic backtracking. This can be done by finding specific structures in a transition diagram of an automaton [2, 18, 29, 34, 36, 37]. Besides, dynamic analysis, such as fuzzing [31], and combinations of static and dynamic analyses [19] are studied. ReDoS repair is a problem of modifying a given regex so that it does not cause ReDoS. Known solutions include exploring ReDoS-free regexes using SMT solvers [6, 21] and rule-based rewriting of vulnerable regexes [20]. These ReDoS detection and repair measures are computationally demanding, and their real-world deployment is limited.

There are other implementation-level studies on speeding up regex matching, such as Just-in-Time (JIT) compilation [17] and FPGA [32]. However, these studies are not intended to prevent catastrophic backtracking.

2 Preliminaries

We introduce preliminaries for this paper. Firstly, we present some basic concepts such as regexes, NFAs, conversion from regexes to NFAs, and backtracking matching. We then discuss *catastrophic backtracking* and the *ReDoS vulnerability* that it can cause. Finally, we introduce *look-around* and *atomic grouping* as practical regex extensions and *NFAs with sub-automata* for these extensions.

We fix a finite set Σ as an alphabet throughout this paper. We call sequences of elements of Σ *strings*. The *empty string* is denoted by ε. For a string $w = \sigma_0\sigma_1 \dots \sigma_{n-1}$, the *length* of w, denoted by $|w|$, is defined as $|w| = n$. We also write $w[i] = \sigma_i$ for $i \in \{0, \dots, n-1\}$.

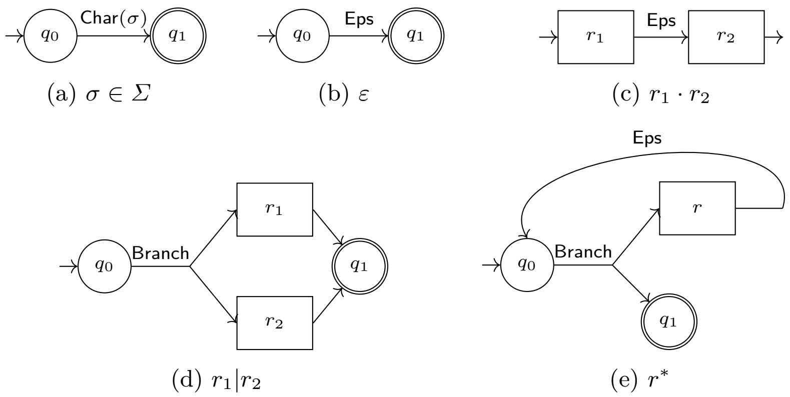
We use partial functions for memoization. For two sets A and B, a partial function G from A to B, denoted by $G\colon A \rightharpoonup B$, is defined as a function $G\colon A \to B \cup \{\bot\}$. Here $\bot$ is the element for "undefined," and it is assumed that $\bot \notin B$.

Let $G\colon A \rightharpoonup B$ be a partial function, $a \in A$, and $b \in B$. We let $G(a) \leftarrow b$ denote an updated partial function: it carries a to b, and any other $x \in A$ to $G(x)$ (it is undefined if $G(x)$ is initially undefined).

2.1 Regexes

Regular expressions (*regexes*) are defined by the following abstract grammar.

$$\begin{array}{rlll} r ::= & \sigma & \text{(a (literal) character, where } \sigma \in \Sigma) & \mid\ \varepsilon \quad \text{(the empty string)} \\ \mid & r|r & \text{(an alternation)} & \mid\ r{\cdot}r \quad \text{(a concatenation)} \\ \mid & r^* & \text{(a repetition)} & \end{array}$$



(a) $\sigma \in \Sigma$ (b) ε (c) $r_1 \cdot r_2$

(d) $r_1|r_2$ (e) r^*

Fig. 1: a conversion from regexes to NFAs

The concatenation operator $\cdot$ may be omitted when there is no ambiguity. The precedence of operators is as follows: repetition, concatenation, and alternation. For example, $ab^*|c$ means $(a \cdot (b^*))|c$.

For a regex r, the *size of* r, denoted by $|r|$, is defined as follows: $|\sigma| = |\varepsilon| = 1$, $|(r_1|r_2)| = |r_1 \cdot r_2| = |r_1| + |r_2| + 1$, and $|r^*| = |r| + 1$.

2.2 NFAs

A *non-deterministic finite state automaton* (*NFA*) is a quadruple (Q, q_0, F, T), where Q is a finite set of states $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of accepting states, and T is a transition function. For each $q \in Q \setminus F$, $T(q)$ can be one of the following: $T(q) = \mathsf{Eps}(q')$, $T(q) = \mathsf{Branch}(q', q'')$, and $T(q) = \mathsf{Char}(\sigma, q')$ where $q', q'' \in Q$ and $\sigma \in \Sigma$.

The above definition of a transition function T is tailored to our purpose of backtracking. Compared to the common definition $\delta\colon Q \times (\{\varepsilon\} \cup \Sigma) \to 2^Q$, it expresses general branching as combinations of certain *elementary branchings*. The latter is namely one transition by ε, two transitions by ε, and one transition by a certain character $\sigma \in \Sigma$. This makes the description of backtracking matching easier. Note, in particular, that the successors q', q'' in the branching $\mathsf{Branch}(q', q'')$ are ordered. Here, q' and q'' are called the *first* and *second successors*, respectively. This definition of transition functions is similar to the op-codes of many real-world regex-matching implementations (cf. [8]).

We present a conversion from regexes to NFAs (see Figure 1); it is similar to the Thompson–McNaughton–Yamada construction [23, 35]. For a regex r, $\mathcal{A}(r)$ denotes the *NFA* $\mathcal{A}$ *converted from* r. In the figure, labels on arrows show kinds of transitions. In a Branch transition, the top arrow points to the first successor, and the bottom points to the second successor. Rectangles indicate that the conversion is applied to sub-expressions inductively. Because each case of this construction introduces at most two new states, for a regex r and the NFA $\mathcal{A}(r) = (Q, q_0, F, T)$, we have $|Q| = O(|r|)$.

Algorithm 1 a partial backtracking matching algorithm for NFAs

```
1: function $\textsc{Match}_{\mathcal{A},w}(q, i)$
       Parameters: an NFA $\mathcal{A}$, and an input string $w$
       Input: a current state $q$, and a current position $i$
       Output: returns $\mathsf{SuccessAt}(i')$ if the matching succeeds, or
               returns $\mathsf{Failure}$ if the matching fails
2:     $(Q, q_0, F, T) = \mathcal{A}$
3:     if $q \in F$ then return $\mathsf{SuccessAt}(i)$
4:     else if $T(q) = \mathsf{Eps}(q')$ then return $\textsc{Match}_{\mathcal{A},w}(q', i)$
5:     else if $T(q) = \mathsf{Branch}(q', q'')$ then
6:         result ← $\textsc{Match}_{\mathcal{A},w}(q', i)$
7:         if result = Failure then return $\textsc{Match}_{\mathcal{A},w}(q'', i)$
8:         else return result
9:     else if $T(q) = \mathsf{Char}(\sigma, q')$ then
10:        if $i < |w|$ and $w[i] = \sigma$ then return $\textsc{Match}_{\mathcal{A},w}(q', i + 1)$
11:        else return Failure
```

We collectively call Eps and Branch transitions *ε-transitions.* Later in this paper, if there are consecutive ε-transitions, they may be shown as a single transition in a figure. When a certain state returns to itself by ε-transitions, such a sequence of ε-transitions is called an *ε-loop.* ε-loops are problematic in matching because they cause infinite loops in matching.

An ε-loop can be detected during matching by recording a position on an input string when a state is visited. When an ε-loop is detected, several solutions exist to deal with it (see, e.g., [30]), such as treating an ε-loop as a failure (e.g., JavaScript and RE2) or treating it as a success but escaping it (e.g., Perl). These solutions can be easily adapted to our algorithms; therefore, for the simplicity of presentation, we introduce the following assumption.

Assumption 1 (no ε-loops). NFAs do not contain ε-loops.

2.3 Backtracking Matching

We present a basic backtracking matching algorithm for NFAs in Algorithm 1. It serves as a basis for optimization by memoization, both in [10] and in the current work.

The function $\textsc{Match}_{\mathcal{A},w}$ is recursively called in this algorithm, but it must terminate on Asm. 1. It takes two parameters: $\mathcal{A}$ is an NFA, and w is an input string. It also takes two arguments: $q \in Q$ is the current state, and $i \in \{0, \ldots, |w|\}$ is the current position on w. $\textsc{Match}_{\mathcal{A},w}(q_0, i)$ for an NFA $\mathcal{A} = (Q, q_0, F, T)$ returns $\mathsf{SuccessAt}(i')$ with the matching position $i' \in \{0, \ldots, |w|\}$ if the matching with $\mathcal{A}$ succeeds from i to i' on w, or returns $\mathsf{Failure}$ if the matching fails.

The $\textsc{Match}$ function implements *partial matching*: given the position $i \in \{0, \ldots, |w|\}$ of interest, one obtains, by running $\textsc{Match}_{\mathcal{A},w}(q_0, i)$, one "matching position" i' (if it exists) such that $w[i]\, w[i{+}1] \ldots w[i']$ is accepted by $\mathcal{A}$. Note the

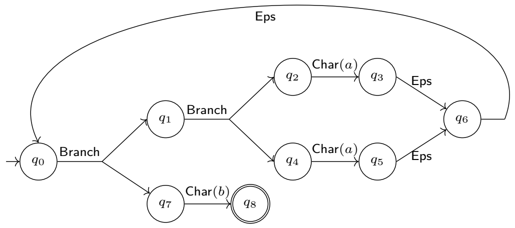


Fig. 2: the NFA $\mathcal{A}((a|a)^*b)$

difference from *total matching*: given $\mathcal{A}$ and w, it returns **true** if (the whole) w is accepted by $\mathcal{A}$ and **false** otherwise. The practical relevance of partial matching must be clear, as we can use it for text search and replacement.

Lines 5 to 8 in Algorithm 1 perform matching for Branch transitions. Here, the algorithm first tries matching from the first successor q', and if that fails, it tries matching from the second successor q'' with the same position. This behavior is called *backtracking*.

We define the *regex partial matching* problem using the function MATCH.

Problem 1 (regex partial matching).
Input: a regex r, an input string w, and a starting position $i \in \{0, \dots, |w|\}$
Output: returns $\textsc{Match}_{\mathcal{A}(r),w}(q_0, i)$ where $\mathcal{A}(r) = (Q, q_0, F, T)$.

Remark 1. One can say that the problem formulation is a bit strange. It requires, as output, a specific matching position chosen by a specific algorithm MATCH, while a usual formulation would require an arbitrary matching position. We take this formulation since we aim to show that our optimization by memoization not only solves partial matching but also is *consistent* with an existing backtracking matching algorithm, in the sense we discussed in §1. We formulate consistency as correctness with respect to Prob. 1, that is, preserving the solution chosen by the specific algorithm MATCH. We also note that the algorithm MATCH mirrors many existing implementations of regex matching (cf. §2.2).

2.4 Catastrophic Backtracking and ReDoS

In the execution of the MATCH function (Algorithm 1), depending on an NFA $\mathcal{A}$ and an input string w, the number of recursive calls for the MATCH function may increase explosively, resulting in a very long matching time, as we will see in Example 1. This explosive increase in matching time is called *catastrophic backtracking*.

Example 1 (catastrophic backtracking). Consider the NFA $\mathcal{A} = \mathcal{A}((a|a)^*b) = (Q, q_0, F, T)$ shown in Figure 2, and let $w =$ "$a^n c$" (the string repeating a of n times and ending with c) be an input string. $\textsc{Match}_{\mathcal{A},w}(q_0, 0)$ invokes recursive calls $O(2^n)$ times until returning Failure. The reason for this recursive call explosion is to try all combinations on q_2 to q_3 and q_4 to q_5 transitions for each a in w during the matching.

Regexes denial of service (*ReDoS*) is a security vulnerability caused by catastrophic backtracking. In ReDoS, catastrophic backtracking causes a huge load on servers, making them unable to respond in a timely manner. There are cases of service outages due to ReDoS at Stack Overflow in 2016 [12] and at Cloudflare in 2019 [16]. Additionally, a 2018 study [33] reported that over 300 web services have potential ReDoS vulnerabilities. Thus, ReDoS is a widespread problem in the real world, and there is a great need for countermeasures.

According to a 2019 study [25], only 38% of developers are aware of ReDoS. This study also found that many developers find it difficult not only to read regexes but also to find and validate regexes to match their desires. It is mentioned in [25] that developers use Internet resources such as Stack Overflow to find regexes. In recent years, it has also become common to use generative AIs such as ChatGPT for such a purpose. However, when the authors asked, "Please suggest 10 regexes for validating email addresses" to ChatGPT,[6] 2 of the 10 suggested regexes would cause ReDoS (see Table 1). Developers may unknowingly use such vulnerable regexes. For this reason, it is important to develop ReDoS countermeasures that can be achieved without the developer being aware of them.

Matching speed-up is a way to avoid causing ReDoS by ensuring that matching is linear in time to the length of an input string, freeing developers from worrying about ReDoS. A popular method for matching speed-up is using breadth-first search for non-deterministic transition instead of backtracking (depth-first search); it is called the *on-the-fly DFA construction* [7,28]. However, since look-around and atomic grouping are extensions based on backtracking (see §2.6), it is not obvious that they can be supported by the on-the-fly DFA construction.

Memoization is another approach to ensuring linear-time backtracking matching; we pursue it in this paper.

2.5 Regex Extensions: Look-around and Atomic Grouping

Many real-world regexes come with various extensions for enhanced expressivity [13]. In this paper, we are interested in two classes of extensions, namely *look-around* and *atomic grouping.*

Look-around Look-around is a regex extension that allows constraints on strings around a certain position. It is also called *zero-width assertion* (e.g., in [10]) because it does not consume any characters. Look-around consists of four types: *positive* or *negative*, and *look-ahead* or *look-behind.*

Positive look-ahead is typically represented by the syntax `(?=`r`)`; its matching succeeds when, reading ahead from the current position of the input string, the

[6] We used ChatGPT 3.5 (September 25, 2023 version).

[7] The second and third regexes are the same; they are the actual output of ChatGPT.

Table 1: the regexes given by ChatGPT for the question "Please suggest 10 regexes for validating email addresses".[7]

suggested regex (ChatGPT's comment)	vulnerable?
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` (Basic Email Validation)	
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$` (Basic Email Validation with TLD)	
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$` (Strict Email Validation)	
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}(?:\.[a-zA-Z]{2,})?$` (Email Validation Allowing for Subdomains)	
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{1,}$` (Email Validation Allowing Single-Character Domain Name)	
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}(?:\.[\p{L}\p{N}]{2,})?$` (Email Validation Allowing Internationalized Domain Names (IDNs))	
`^(?:"[\w\s]+")?([a-zA-Z0-9._%+-]+)@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` (Email Validation with Optional Quoted Local Part)	
`^(?:\([^()]*\)|[\w\s]+)?([a-zA-Z0-9._%+-]+)@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` (Email Validation with Optional Comments)	vulnerable
`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$i` (Email Validation Allowing for Case-Insensitive Domain)	
`^(?:(?:[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})|([a-zA-Z0-9._%+-]+)\+[^@]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})$` (Email Validation with Support for Subaddressing)	vulnerable

matching of the inner regex r succeeds. Note that the position for the overall matching does not change by the inner matching of r. For example, the regex `/(?=bc)/` matches the string `"abc"` from position 1 (i.e., after the first character `a`) without consuming any characters.

The matching of a negative look-ahead `(?!`r`)` succeeds when the inner regex r is *not* matched.

Positive or negative *look-behind*—denoted by `(?<=`r`)` or `(?<!`r`)`, respectively—is similar to the above, with the difference that the inner matching of r is performed *backward*, i.e., from right to left. For example, the regex `/(?<=ab)/` matches the string `"abc"` from position 2 (i.e., before the last character `c`) without consuming any characters.

A typical use of look-around is to put a look-behind before (or a look-ahead after) a regex r. This is useful when one wants to perform a search or replacement of r for only those occurrences that are in a certain context. For example, the regex `/(?<=<p>)[^<]*(?=<\/p>)/` matches only contents of the HTML `<p>` tag. As another example, common assertions such as `\A` (this matches the beginning of a string) and `\z` (this matches the end) can be expressed using look-around, namely `\A` = `(?<!.)` and `\z` = `(?!.)`.

Atomic Grouping Atomic grouping is a regex extension that controls backtracking behaviors. It is designed to manually avoid problems caused by backtracking, such as catastrophic backtracking (§2.4).

Atomic grouping is represented by the syntax `(?>`r`)`; once the matching of the inner regex r succeeds, the remaining branches in potential backtracking for matching r are discarded. For example, the regex `/(a|ab)c/` matches the string `"abc"`, but the regex `/(?>(a|ab))c/` using atomic grouping does *not* match it. This is because, once `a` in the atomic grouping matches the first character `a` of `"abc"`, the remaining branch `ab` (in `a|ab`) is discarded, and one is left with the regex `c` and the string `"bc"`.

Atomic grouping is often used for the purpose of preventing catastrophic backtracking. In that case, it is used in combination with the repetition syntax, e.g., `(?>(`r`*))` (often abbreviated as r`*+`) and `(?>(`r`+))` (abbrev. as r`++`). These abbreviations are called *possessive quantifiers*. The former (namely `(?>(`r`*))`) is intuitively understood as `(?>(`ε`|`r`|`rr`|...))`, with the difference that longer matching is preferred (this is because the `Eps` loop is the first successor in Figure 1e). Once a longer match is found, the remaining branches (i.e., those for shorter matches) get discarded, thus preventing catastrophic backtracking.

One might wonder if our (linear-time and thus ReDoS-free) matching algorithm should support atomic grouping—the principal use of atomic grouping is to suppress backtracking and avoid ReDoS. We do need to support it since, as we discussed in §1, ours is meant to be a drop-in replacement for matching implementations that are currently used.

Our Target Extended Regexes Our target class, namely *regexes with look-around and atomic grouping*, is defined by the following grammar.

$$
\begin{array}{lll}
r ::= & \ldots & \text{(the same as the regexes definition, §2.1)} \\
 & \mid \texttt{(?=}r\texttt{)} \mid \texttt{(?!}r\texttt{)} & \text{(positive and negative look-ahead)} \\
 & \mid \texttt{(?<=}r\texttt{)} \mid \texttt{(?<!}r\texttt{)} & \text{(positive and negative look-behind)} \\
 & \mid \texttt{(?>}r\texttt{)} & \text{(atomic grouping)}
\end{array}
$$

For brevity, we sometimes refer to regexes with look-around and atomic grouping as *(la, at)-regexes*. We also refer to regexes with look-around as *la-regexes* and regexes with atomic grouping as *at-regexes*.

For a (la, at)-regex r, the size of r, denoted by $|r|$, is defined as the same as the regex one except for $|\texttt{(?=}r\texttt{)}| = |\texttt{(?>}r\texttt{)}| = |r| + 1$.

Look-around is known to be *regular*: they can be converted to DFA, and the language family of la-regexes is the same as the regular language. This fact is mentioned in [3,26,27]. Atomic grouping is also known to be regular in the same sense [4]. However, it is known that look-ahead and atomic grouping can make the number of states of the corresponding DFA grow exponentially [3,4,26,27].

In what follows, for simplicity, we only discuss positive look-ahead in discussions of look-around. Adaptation to other look-around operators, such as negative look-behind, is straightforward.

2.6 NFAs with Sub-automata

We introduce *NFAs with sub-automata* for backtracking matching algorithms for (la, at)-regexes. This extended notion of NFAs is suggested in [10, Section IX.B], but it seems ours is the first formal exposition.

Roughly speaking, an NFA with sub-automata is an NFA whose transitions can be labeled with—in addition to a character $\sigma \in \Sigma$, as in usual NFAs—another NFA with sub-automata. See Figure 3, where transitions from q_0 to q_1 are labeled with $\boxed{r}$, the NFA with sub-automata obtained by converting r. We annotate these transitions further with a label (pla for positive look-ahead, at for atomic grouping, etc.) that indicates which operator they arise from. Note that NFAs with sub-automata can be nested—transitions in $\boxed{r}$ in Figure 3 can be labeled with NFAs with sub-automata, too.

Our precise definition is as follows. There, P is the set that collects all states that occur in an NFA with sub-automata $\mathcal{A}$, i.e., in 1) the top-level NFA, 2) its label NFAs, 3) their label NFAs, and so on.

Definition 1 (NFAs with sub-automata). *An* NFA with sub-automata $\mathcal{A}$ *is a quintuple $\mathcal{A} = (P, Q, q_0, F, T)$ where P is a finite set of states and $Q \subseteq P$ is a set of so-called* top-level states. *We require that the quadruple (Q, q_0, F, T) is an NFA, except that the value $T(q)$ of the transition function T is either 1) $\mathsf{Eps}(q')$, $\mathsf{Branch}(q', q'')$, or $\mathsf{Char}(\sigma, q')$ (as in usual NFAs, §2.2), or 2) $\mathsf{Sub}(k, \mathcal{A}', q')$, where $\mathcal{A}'$ is an NFA with sub-automata, q' is a successor state, and k is a* kind label *where $k \in \{\mathsf{pla}, \mathsf{nla}, \mathsf{plb}, \mathsf{nlb}, \mathsf{at}\}$.*

We further impose the following requirements. Firstly, we require all NFAs with sub-automata in $\mathcal{A}$ to have disjoint state spaces. That is, for any distinct top-level states $q, q'' \in Q$ in $\mathcal{A}$, if $T(q) = \mathsf{Sub}(k, \mathcal{A}', q')$ and $T(q'') = \mathsf{Sub}(k', \mathcal{A}'', q''')$, then we must have $P' \cap P'' = \emptyset$, $Q \cap P' = \emptyset$ and $Q \cap P'' = \emptyset$, where $\mathcal{A}' = (P', \dots)$ and $\mathcal{A}'' = (P'', \dots)$. Secondly, we require that the set P in $\mathcal{A} = (P, \dots)$ is the (disjoint) union of all states that occur within $\mathcal{A}$, that is, $P = Q \cup \bigcup_{q \in Q, T(q) = \mathsf{Sub}(k, \mathcal{A}', q'), \mathcal{A}' = (P', \dots)} P'$.

The kind label k in $\mathsf{Sub}(k, \mathcal{A}', q'')$ indicates how the sub-automaton $\mathcal{A}'$ should be used (cf. Algorithm 2). If every kind label occurring in $\mathcal{A}$ (including its sub-automata) is either $\mathsf{pla}, \mathsf{nla}, \mathsf{plb}$, or nlb, then $\mathcal{A}$ is called a *la-NFA*. Similarly, if every kind label is at, $\mathcal{A}$ is called an *at-NFA*. Following this convention, general NFAs with sub-automata are called *(la, at)-NFAs.*

Note that the definition is recursive. Non-well-founded nesting is prohibited, however, by the finiteness of P. By the definition, if $P = Q$, then $\mathcal{A}$ does not contain any transitions labeled with sub-automata.

In addition to Eps and Branch transitions, we refer to Sub transitions with a label $k \in \{\mathsf{pla}, \mathsf{nla}, \mathsf{plb}, \mathsf{nlb}\}$ as ε-transitions too. We also assume the following, similarly to Asm. 1.

Assumption 2. (la, at)-NFAs do not contain ε-loops.

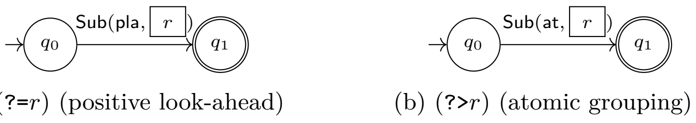


Fig. 3: a conversion from (la, at)-regexes to (la, at)-NFAs. For negative look-ahead, we use the corresponding kind label nla. For positive and negative look-behind, besides using the kind labels plb and nlb, we suitably reverse $\boxed{r}$.

Algorithm 2 a partial backtracking matching algorithm for (la, at)-NFAs

```
1: function $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A},w}(q, i)$
         Parameters: a (la, at)-NFA $\mathcal{A}$, and an input string $w$
              Input: a current state $q$, and a current position $i$
             Output: returns SuccessAt($i'$) if the matching succeeds, or
                     returns Failure if the matching fails
2:     $(P, Q, q_0, F, T) = \mathcal{A}$
3:     if $q \in F$ then
           ⋮                                              ▷ the same as Algorithm 1
12:    else if $T(q) = \mathsf{Sub}(\mathsf{pla}, \mathcal{A}', q')$ then
                               ▷ positive look-ahead; other look-around ops. are similar
13:        $(P', Q', q_0', F', T') = \mathcal{A}'$
14:        result ← $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A}',w}(q_0', i)$
15:        if result = SuccessAt($i'$) then return $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A},w}(q', i)$
16:        else return $r$
17:    else if $T(q) = \mathsf{Sub}(\mathsf{at}, \mathcal{A}', q')$ then               ▷ atomic grouping
18:        $(P', Q', q_0', F', T') = \mathcal{A}'$
19:        result ← $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A}',w}(q_0', i)$
20:        if result = SuccessAt($i'$) then return $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A},w}(q', i')$
21:        else return $r$
```

For (la, at)-regexes, their conversion to (la, at)-NFAs is described by the constructions in Figure 3—using transitions labeled with sub-automata—in addition to the conversion for regexes in §2.2. Note that we have $|P| = O(|r|)$ in these constructions.

The backtracking matching algorithm in Algorithm 1 can be naturally extended to (la, at)-NFAs; it is shown in Algorithm 2. The clauses for positive look-ahead (Lines 12 to 16) and atomic grouping (Lines 17 to 21) are similar to each other, conducting matching for sub-automata. Note that their difference is in the "return position" (i in Line 15; i' in Line 20).

The clauses for other look-around operators are similar to the ones for positive look-around. For look-behind, we can suitably use an additional parameter $d \in \{-1, +1\}$ for indicating a matching direction.

Using the extended backtracking matching algorithm (Algorithm 2), we define the *partial matching problem for (la, at)-regexes* in the same way as for regexes without extensions (Prob. 1).

Problem 2 ((la, at)-regex partial matching).
Input: a (la, at)-regex r, an input string w, and a starting position i
Output: returns $\textsc{Match}\text{-}(\mathrm{la},\mathrm{at})_{\mathcal{A}(r),w}(q_0, i)$ where $\mathcal{A}(r) = (P, Q, q_0, F, T)$.

3 Previous Works on Regex Matching with Memoization

This section introduces an existing work [10] on regex matching with memoization, paving the way for our algorithms for (la, at)-regexes in Sections 4 and 5.

Memoization is a programming technique that makes recursive computations more efficient by 1) recording arguments of a function and the corresponding return values and 2) reusing them when the function is called with the recorded arguments.

As we described in §2.3, regex matching is conducted by backtracking matching. It is implemented by recursive functions (see Algorithms 1 and 2); thus, it is a natural idea to apply memoization. Since Java 14, Java's regex implementation has indeed used memoization for optimization. However, this optimization is not enough to completely prevent ReDoS; see, e.g., [24].

The work that inspires the current work the most is [10], whose main novelty is linear-time backtracking regex matching (much like the current work). Its contributions are as follows.

1. Focusing on (non-extended) regexes (see §2.1), they introduce a backtracking matching algorithm that uses memoization. It achieves a linear-time complexity: for an input string w, its runtime is $O(|w|)$.
2. They introduce *selective memoization*, by which they reduce the domain of the memoization table from $Q \times \mathbb{N}$ to $Q_{\mathsf{sel}} \times \mathbb{N}$. Here Q_{sel} is a subset of Q that is often much smaller.
3. They introduce a memory-efficient compression method—based on run-length encoding (RLE)—for memoization tables.
4. Finally, they discuss adaptations of the above method to extended regexes, namely REWZWA (the extension by look-around; look-around is called *zero-width assertion* in [10]) and REWBR (the extension by *back-reference*).

We will mainly discuss the above item 1; it serves as a basis for our algorithms in Sections 4 and 5. The technique in the item 2 is potentially very relevant: we expect that it can be combined with the current work; doing so is future work. The content of the item 2 is reviewed in [15, Appendix A] for the record.

Remark 2. On the above item 4, the work [10] claims that the time complexity of their algorithm is linear also for REWZWA ($O(|w|)$ for an input string w). However, we believe that this claim comes with the following problems.

– The description of an algorithm for REZWA in [10] is abstract and leaves room for interpretation. The description is to "preserve the memo functions of the sub-automata throughout the simulation of the top-level M-NFA, remembering the results from sub-simulations that begin at different indices i of w" [10, Section IX-B]. For example, it is not explicit what the "results" are—they can mean (complete) matching results or mere success/failure.

Algorithm 3 a total matching algorithm with memoization for NFAs without ε-transitions [10, Listing 2].

```
1: function $\textsc{DavisSL}^{M}_{\mathcal{A},w}(q, i)$
       Parameters: an NFA $\mathcal{A}$ without $\varepsilon$-transitions, an input string $w$, and
                   a memoization table $M\colon Q \times \mathbb{N} \rightharpoonup \{\mathbf{false}\}$
            Input: a current state $q$, and a current position $i$
           Output: returns true if the matching succeeds, or
                   returns false if the matching fails
2:     $(Q, q_0, F, \delta) = \mathcal{A}$
3:     if $i = |w|$ then return whether $q \in F$
4:     if $M(q, i) \neq \bot$ then return $M(q, i)$
5:     for $q' \in \delta(q, w[i])$ do
6:         if $\textsc{DavisSL}^{M}_{\mathcal{A},w}(q', i + 1)$ then return true
7:     $M(q, i) \leftarrow$ false
8:     return false
```

– Moreover, the part "that begin at different indices i of w" is problematic; we believe that remembering these results does not lead to linear-time complexity. This point is discussed later in Remark 4.
– Besides, there is a gap between the algorithm described in the paper [10] and its prototype implementation [11], even for (non-extended) regexes. See Remark 3.
– Because of this gap, the implementation [11] works in linear time for all regexes, including REZWA, but can lead to erroneous results for REZWA. See Remark 4.

Our contribution includes a correct memoization algorithm for look-around (REZWA) that resolves the above problems.

3.1 Linear-time Backtracking Matching with Memoization

We describe the first main contribution of the work [10] (the item 1 in the above list), namely a backtracking algorithm that achieves a linear-time complexity thanks to memoization. The algorithm [10, Listing 2] is presented in Algorithm 3.

In this algorithm $\textsc{DavisSL}^{M}_{\mathcal{A},w}$, an NFA $\mathcal{A}$ is a quintuple (Q, q_0, F, δ) where $\delta\colon Q \times \Sigma \to 2^Q$ is an nondeterministic transition function. An additional parameter $M\colon Q \times \mathbb{N} \rightharpoonup \{\mathbf{false}\}$ is a memoization table, which is mathematically a mutable partial function. This algorithm implements total matching (cf. §2.3). It is notable that the memoization table records only matching *failures*: a matching *success* does not have to be recorded since it immediately propagates to the success of the whole problem.

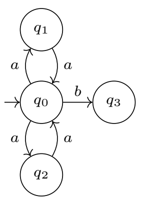


Fig. 4: the NFA $\mathcal{A}((aa|aa)^*b)$, after removing ε-transitions

Algorithm 4 a variant of Algorithm 3 implemented in their prototype [11]

```
1: function $\textsc{DavisSLImpl}^{M}_{\mathcal{A},w}(q,i)$
2:     $(Q, q_0, F, \delta) = \mathcal{A}$
3:     if $i = |w|$ then return whether $q \in F$
4:     if $M(q,i) \neq \bot$ then return $M(q,i)$
       $M(q,i) \leftarrow$ false                          ▷ $M(q,i)$ is speculatively set to false
5:     for $q' \in \delta(q, w[i])$ do
           ⋮
7:     ~~$M(q,i) \leftarrow$ false~~                           ▷ moved up
       ⋮
```

This algorithm achieves a linear-time matching. It thus prevents ReDoS. A full proof of linear-time complexity is found in [10, Appendix C], but its essence is the following (note the critical role of memoization here).

– For any call $\textsc{DavisSL}^{M}_{\mathcal{A},w}(q,i)$, if $M(q,i)$ is defined, then the call does not invoke any further recursive calls.
– When such a call returns **false**, the entry $M(q,i)$ of the memoization gets defined (Line 7).
– As a consequence, the number of recursive calls of $\textsc{DavisSL}^{M}_{\mathcal{A},w}$ is limited to $|Q| \times |w|$.

Example 2 (matching with memoization for NFAs without ε-transitions). Let us consider the regex $(aa|aa)^*b$ and the corresponding NFA $\mathcal{A}((aa|aa)^*b)$ defined in §2.2. For the purpose of applying Algorithm 3, we manually remove its ε-transitions, leading to the NFA in Figure 4. Let $w =$ "$a^{2n}c$" be an input string. $\textsc{Match}_{\mathcal{A},w}(q_0, 0)$ (without memoization) invokes recursive calls $O(2^n)$ times for the same reason as in Example 1, but $\textsc{DavisSL}^{M_0}_{\mathcal{A},w}(q_0, 0)$ (with memoization, where M_0 is the initial memoization table) invokes recursive calls $O(n)$ times because $M(q_0, i)$ for each position $i \in \{0, 2, \ldots, 2n\}$ has been recorded after the first visit.

Remark 3. Following the discussion in Remark 2, here we describe the gap between Algorithm 3—the algorithm described in the paper [10]—and its prototype implementation [11]. The latter is shown in Algorithm 4.

The precise difference between the two algorithms is that Line 7 in Algorithm 3 is moved up to the moment just before the for-loop, in Algorithm 4. It is not hard to see that this modification does not affect the correctness of the algorithm: if the pair (q,i) is visited again in the future, it means that the current matching from (q,i) did not succeed, and backtracking occurred. Note that, in case the current matching is successful, the function call returns **true** so the memoization content $M(q,i)$ should not matter.

However, the above argument is true only when there is no look-around. (A detailed discussion is in Example 3.) This point seems to be missed in the implementation [11].

Algorithm 5 a partial matching algorithm with memoization. An adaptation of Algorithm 3 from [10], and a basis for our algorithms (Algorithms 6 and 7)

```
1: function $\textsc{Memo}^{M}_{\mathcal{A},w}(q,i)$
         Parameters: an NFA $\mathcal{A}$, an input string $w$, and
                     a memoization table $M\colon Q \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}\}$
              Input: a current state $q$, and a current position $i$
             Output: returns $\mathsf{SuccessAt}(i')$ if the matching succeeds, or
                     returns $\mathsf{Failure}$ if the matching fails
2:     $(Q, q_0, F, T) = \mathcal{A}$
3:     if $M(q,i) \neq \bot$ then return $M(q,i)$
4:     result $\leftarrow \bot$
5:     if $q \in F$ then result $\leftarrow \mathsf{SuccessAt}(i)$
6:     else if $T(q) = \mathsf{Eps}(q')$ then result $\leftarrow \textsc{Memo}^{M}_{\mathcal{A},w}(q',i)$
7:     else if $T(q) = \mathsf{Branch}(q',q'')$ then
8:         result $\leftarrow \textsc{Memo}^{M}_{\mathcal{A},w}(q',i)$
9:         if result = Failure then result $\leftarrow \textsc{Memo}^{M}_{\mathcal{A},w}(q'',i)$
10:    else if $T(q) = \mathsf{Char}(\sigma,q')$ then
11:        if $i < |w|$ and $w[i] = \sigma$ then result $\leftarrow \textsc{Memo}^{M}_{\mathcal{A},w}(q',i+1)$
12:        else result $\leftarrow$ Failure
13:    if result = Failure then $M(q,i) \leftarrow$ Failure
14:    return result                          ▷ result $\neq \bot$, as one can easily see
```

3.2 Matching with Memoization Adapted to the Current Formalism

In Algorithm 5, we present an adaptation of Algorithm 3 to our formalism, especially our definition of NFA (§2.2) that offers fine-grained handling of non-determinism. Algorithm 5 has been adapted also to solve *partial* matching (it returns a matching position i') rather than total matching as in Algorithm 3 (cf. §2.3). Algorithm 5 serves as a basis towards our extensions to look-around and atomic grouping in Sections 4 to 5.

The adaptation is straightforward: Line 5 ensures that the algorithm solves partial matching; the rest is a natural adaptation of the for-loop of Algorithm 3 to our definition of NFA (§2.2). The algorithm terminates thanks to Asm. 1. We note that the type of memoization tables does not have to be changed compared to Algorithm 3.

Algorithm 5 exhibits the same desired properties as Algorithm 3, namely correctness (with respect to Prob. 1) and linear-time complexity. We formally state these properties for the record; here, $M_0\colon Q \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}\}$ is the initial memoization table (its entry is anywhere $\bot$).

Theorem 1 (linear-time complexity of Algorithm 5). *For an NFA $\mathcal{A} = (Q, q_0, F, T)$, an input string w, and an position $i \in \{0, \dots, |w|\}$, $\textsc{Memo}^{M_0}_{\mathcal{A},w}(q_0, i)$ terminates with $O(|w|)$ recursive calls.*

Theorem 2 (correctness of Algorithm 5). *For an NFA $\mathcal{A} = (Q, q_0, F, T)$, an input string w, and an position $i \in \{0, \dots, |w|\}$, $\textsc{Match}_{\mathcal{A},w}(q_0, i) = \textsc{Memo}^{M_0}_{\mathcal{A},w}(q_0, i)$.*

The proofs can be found in [15, Appendix B.1]. Here is their outline.

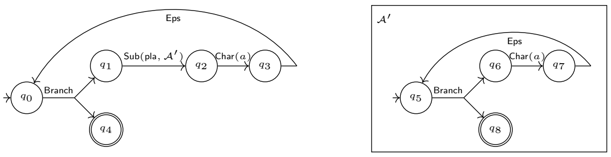


Fig. 5: the la-NFA $\mathcal{A}(((\texttt{?=}a^*)a)^*)$

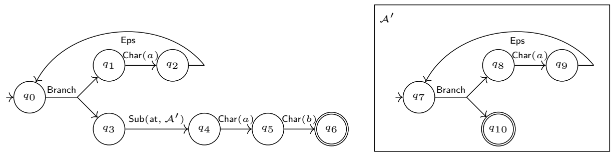


Fig. 6: the at-NFA $\mathcal{A}(a^*(\texttt{?>}a^*)ab)$

We first introduce the notion of run for MATCH and MEMO; it records *recursive calls* of the function itself, as well as *invocations* of the memoization table, together with their return values.

For linear time complexity (Thm. 1), we show that 1) a recursive call with the same argument (q, i) appears at most once in a run, and that 2) the number of invocations of the memoization table with the same key (q, i) is bounded by the (graph-theoretic) in-degree. Linear-time complexity then follows easily.

For correctness (Thm. 2), we introduce a conversion from runs of MEMO to runs of MATCH. By showing that 1) the result is indeed a valid run of MATCH and 2) the conversion preserves return values, we show the coincidence of the return values of the two algorithms, i.e., correctness.

4 Memoization for Regexes with Look-around

We describe our first main technical contribution, namely a backtracking matching algorithm for la-NFAs with memoization (Algorithm 6). We prove that it is correct (Thm. 4) and that its time complexity is linear ($O(|w|)$, Thm. 3).

The key ingredient of our algorithm is the type of memoization tables, where their range is extended from $\{\mathsf{Failure}\}$ to $\{\mathsf{Failure}, \mathsf{Success}\}$. We motivate this extension through two problematic algorithms MEMOEXIT-la and MEMOENTER-la; MEMOEXIT-la is obtained by naively extending Algorithm 5 (MEMO) with adding the processing of sub-automaton transitions with pla (positive look-ahead) done in Algorithm 2 (Lines 12 to 16), and MEMOENTER-la is similar to MEMOEXIT-la, but this records to the memoization table at the same timing as Algorithm 4 (DAVISSLIMPL). In particular, their memoization tables only record false.

The example below shows the problems with the two naive algorithms. Specifically, MEMOEXIT-la is not linear and MEMOENTER-la is not correct.

Example 3. Consider the la-NFA $\mathcal{A} = \mathcal{A}(((\texttt{?=}a^*)a)^*) = (P, Q, q_0, F, T)$ shown in Figure 5, and let $w = \texttt{"}a^n\texttt{"}$ be an input string.

MemoExit-la$^{M_0}_{\mathcal{A},w}(q_0, 0)$ invokes recursive calls $O(|w|^2)$ times—in the same way as Match-$(\mathrm{la}, \mathrm{at})$—because there are no matching failures in $\mathcal{A}'$ that contribute to memoization.

We also see MemoEnter-la is not correct: Match-$(\mathrm{la}, \mathrm{at})_{\mathcal{A},w}(q_0, 0)$ returns $\mathsf{SuccessAt}(n)$, but MemoEnter-la$^{M_0}_{\mathcal{A},w}(q_0, 0)$ returns $\mathsf{SuccessAt}(1)$ because $M(q_5, 1) = \textbf{false}$ is recorded during the first loop and interpreted as a matching failure.

In Example 3, a natural solution to the non-linearity issues with MemoExit-la is to enrich memoization so that it also records previous successes of look-around. Furthermore, since matching positions do not matter in look-around, the type of memoization tables should be $M : P \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}, \mathsf{Success}\}$.

Remark 4. The work [10, Section IX-B] proposes an adaptation of their memoization algorithm to REZWA. Its description in [10, Section IX-B] (to "preserve the memo functions... "; see Remark 2) consists of the following two points:

1. preserving the memoization tables of the sub-automata throughout the whole matching, and
2. recording the results of sub-automata matching from different start positions i of w.

The naive algorithm MemoExit-la we discussed above implements the first point. We can further add the second point (that is essentially "memoization for sub-automaton matching") to MemoExit-la.

However, we find that this is not enough to ensure linear-time complexity. The problem is that the "memoization for sub-automaton matching" is used too infrequently. For example, in Example 3, the start positions of sub-automaton matching are different each time; thus, the memoized results are never used.

Our algorithm (Algorithm 6) resolves this problem by letting the memoization tables (for sub-automaton matching) record results not only for *starting* positions but also for non-starting positions.

We also note that there is a gap between the algorithm in the paper [10] and its prototype implementation [11]; see Remark 3. The latter is linear time but not always correct. For example, in Example 3, the correct result is $\mathsf{SuccessAt}(n)$, but the prototype [11] returns $\mathsf{SuccessAt}(1)$, similarly to MemoEnter-la.

Algorithm 6 is the matching algorithm for la-NFAs that we propose. It adopts the above extended type of M. In Line 18, $\mathsf{Success}$ is recorded in the memoization table when the matching succeeded. This function can return one of $\mathsf{SuccessAt}(i')$, $\mathsf{Failure}$, and $\mathsf{Success}$. We first prove the following lemma to see that the algorithm indeed solves the partial matching problem (Prob. 2).

Lemma 1. *For a la-NFA $\mathcal{A} = (P, Q, q_0, F, T)$, an input string w, and a position $i \in \{0, \dots, |w|\}$, Memo-la$^{M_0}_{\mathcal{A},w}(q_0, i)$ returns either* $\mathsf{SuccessAt}(i')$ *for $i' \in \{0, \dots, |w|\}$ or* $\mathsf{Failure}$ *(it does not return* $\mathsf{Success}$*).*

Proof. When we obtain $\mathsf{Success}$ as a return value, it must be via an entry $M(q, i)$ of the memoization table. However, due to Asm. 2, when $M(q, i)$ is set to $\mathsf{Success}$ for a state q of the top-level automaton of $\mathcal{A}$, the matching is already finished and returns $\mathsf{SuccessAt}(i')$. □

Algorithm 6 our partial matching algorithm with memoization for la-NFAs

```
1: function $\textsc{Memo}\text{-la}^{M}_{\mathcal{A},w}(q,i)$
       Parameters: a la-NFA $\mathcal{A}$, an input string $w$, and
                   a memoization table $M\colon P \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}, \mathsf{Success}\}$
       Input: a current state $q$, and a current position $i$
       Output: returns $\mathsf{SuccessAt}(i')$ if the matching succeeds,
               returns $\mathsf{Success}$ if a matching success is in $M$ (cf. Lemma 1), or
               returns $\mathsf{Failure}$ if the matching fails
2:     $(P, Q, q_0, F, T) = \mathcal{A}$
3:     if $M(q,i) \neq \bot$ then return $M(q,i)$
4:     $\mathsf{result} \leftarrow \bot$
5:     if $q \in F$ then                    ▷ the same as Lines 5 to 12 of Algorithm 5
           ⋮
13:    else if $T(q) = \mathsf{Sub}(\mathsf{pla}, \mathcal{A}', q')$ then
14:        $(P', Q', q_0{}', F', T') = \mathcal{A}'$
15:        $\mathsf{result} \leftarrow \textsc{Memo}\text{-la}^{M}_{\mathcal{A}',w}(q_0{}', i)$
16:        if $\mathsf{result} = \mathsf{SuccessAt}(i')$ or $\mathsf{Success}$ then
17:            $\mathsf{result} \leftarrow \textsc{Memo}\text{-la}^{M}_{\mathcal{A},w}(q', i)$
18:    if $\mathsf{result} = \mathsf{SuccessAt}(i')$ or $\mathsf{Success}$ then $M(q,i) \leftarrow \mathsf{Success}$
19:    else if $\mathsf{result} = \mathsf{Failure}$ then $M(q,i) \leftarrow \mathsf{Failure}$
20:    return $\mathsf{result}$
```

As a consequence of the lemma, we can further shrink the memoization tables in Algorithm 6 by not recording Success for $M(q, i)$ where q is a state of the top-level automaton.

Algorithm 6 exhibits the desired properties, namely correctness (with respect to Prob. 2) and linear-time complexity.

Theorem 3 (linear-time complexity of Algorithm 6). *For a la-NFA $\mathcal{A} = (P, Q, q_0, F, T)$, an input string w, and a position $i \in \{0, \dots, |w|\}$, $\textsc{Memo}\text{-la}^{M_0}_{\mathcal{A},w}(q_0, i)$ terminates with $O(|w|)$ recursive calls.*

Theorem 4 (correctness of Algorithm 6). *For a la-NFA $\mathcal{A} = (P, Q, q_0, F, T)$, an input string w, and a position $i \in \{0, \dots, |w|\}$, $\textsc{Match}\text{-}(\mathrm{la}, \mathrm{at})(q_0, i) = \textsc{Memo}\text{-la}^{M_0}_{\mathcal{A},w}(q_0, i)$.*

Thm. 3 and 4 can be shown similarly to Thm. 1 and 2; see [15, Appendix B.2]. The in-degree for sub-automata requires some additional care.

5 Memoization for Regexes with Atomic Grouping

We describe our second main technical contribution, namely a backtracking matching algorithm for at-NFAs with memoization (Algorithm 7). We prove that it is correct (Thm. 6) and that its time complexity is linear ($O(|w|)$, Thm. 5).

The key ingredient of our algorithm is the type of memoization tables, where their range is extended from $\{\mathsf{Failure}\}$ to $\{\mathsf{Failure}(j) \mid j \in \{0, \dots, \nu(\mathcal{A}_0)\}\}$; the latter records a *depth* j of atomic grouping in order to distinguish failures of different depths. We motivate this extension through two problematic algorithms

MemoExit-at and MemoEnter-at. Much like in §4, MemoExit-at naively extends Algorithm 5 (Memo) by adding the processing of sub-automaton transitions with at done in Algorithm 2 (Lines 17 to 21), and MemoEnter-la is similar to MemoExit-at, but records to the memoization table at the same timing as Algorithm 4 (DavisSLImpl).

Firstly, we observe that MemoExit-at is not linear for a reason similar to Example 3. (A concrete example is given by Example 4.) Therefore, we turn to the other candidate, namely MemoEnter-at.

We find, however, that MemoEnter-at is also problematic. It is not correct.

Example 4. Consider the at-NFA $\mathcal{A} = \mathcal{A}(a^*(\texttt{?>}a^*)ab) = (P, Q, q_0, F, T)$ shown in Figure 6, and let $w = \texttt{"}a^n b\texttt{"}$ be an input string. $\textsc{Match-}(\mathrm{la}, \mathrm{at})_{\mathcal{A},w}(q_0, 0)$ returns $\mathsf{Failure}$—the atomic grouping $(\texttt{?>}a^*)$ consumes all a's in w and no a is left for the final ab pattern—but $\textsc{MemoEnter-at}^{M_0}_{\mathcal{A},w}(q_0, 0)$ returns $\mathsf{SuccessAt}(n+1)$. Thus MemoEnter-at is wrong.

For both algorithms, the state q_7 in the at transition is first reached at position $i = n$, and then backtracking is conducted, leading to the state q_7 again at $i = n - 1$. The execution of MemoEnter-at proceeds as follows.

- The first execution path consumes all a's in the loop from q_0 to q_2, reaches q_7 with $i = n$, eventually leading to failure at q_4 and thus to backtracking. Speculative memoization ($M(q, i) \leftarrow \textbf{false}$ in Algorithm 4) is conducted in its course; in particular, $M(q_7, n) = \textbf{false}$ is recorded.
- After backtracking, the second execution path reaches q_7 with $i = n - 1$; it then visits q_8 once and reaches q_7 with $i = n$. Now it uses the memoized value $M(q_7, n) = \textbf{false}$ (cf. Line 4 of Algorithm 4), leading to backtracking to q_7 with $i = n-1$. It then takes the branch to q_{10}, and the matching for $\mathcal{A}'$ succeeds. Therefore, the execution reaches q_4 with $i = n - 1$, and the whole matching succeeds.

The last example shows the challenge we are facing, namely the need of *distinguishing failures of different depths.* Specifically, in the previous example, the memoized value $M(q_7, n) = \textbf{false}$ comes from the failure of matching for ambient $\mathcal{A}$; still, it is used to control backtracking in the sub-automaton $\mathcal{A}'$. This fact is problematic in an atomic grouping where, roughly speaking, backtracking in an ambient automaton should not cause backtracking in a sub-automaton. Atomic grouping can be nested, so we must track at which depth failure has happened.

Definition 2 (nesting depth of atomic grouping). *For an at-NFA $\mathcal{A} = (P, Q, q_0, F, T)$ and a state $q \in P$, the* nesting depth of atomic grouping for q, *denoted by $\nu_{\mathcal{A}}(q)$, is*

$$\nu_{\mathcal{A}}(q) = \begin{cases} 0 & \textit{if } q \in Q \\ 1 + \nu_{\mathcal{A}'}(q) & \textit{where } \mathcal{A}' = (P', Q', q'_0, F', T') \\ & \textit{s.t. } T(q') = \mathsf{Sub}(\mathsf{at}, \mathcal{A}', q'') \textit{ and } q \in P'. \end{cases}$$

Algorithm 7 our partial matching algorithm with memoization for at-NFAs

```
1: function $\textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q,i)$
       Parameters: an at-NFA $\mathcal{A}_0$, a sub-automaton $\mathcal{A}$ of $\mathcal{A}_0$ (it can be $\mathcal{A}_0$ itself),
                   an input string $w$, and a memoization table $M\colon P \times \mathbb{N} \rightharpoonup$
                        $\{\mathsf{Failure}(j) \mid j \in \{0,\dots,\nu(\mathcal{A}_0)\}\}$
            Input: a current state $q$, and a current position $i$
           Output: returns $\mathsf{SuccessAt}(i',K)$ if the matching succeeds, or
                   returns $\mathsf{Failure}(j)$ if the matching fails
2:     $(P,Q,q_0,F,T) = \mathcal{A}$
3:     if $M(q,i) \neq \bot$ then return $M(q,i)$
4:     $\mathsf{result} \leftarrow \bot$
5:     if $q \in F$ then $\mathsf{result} \leftarrow \mathsf{Success}(i,\emptyset)$
6:     else if $T(q) = \mathsf{Eps}(q')$ then $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q',i)$
7:     else if $T(q) = \mathsf{Branch}(q',q'')$ then
8:         $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q',i)$
9:         if $\mathsf{result} = \mathsf{Failure}(j)$ and $j = \nu_{\mathcal{A}_0}(q)$ then
10:            $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q'',i)$
11:            if $\mathsf{result} = \mathsf{Failure}(j')$ then $\mathsf{result} \leftarrow \mathsf{Failure}(\min(j,j'))$
12:    else if $T(q) = \mathsf{Char}(\sigma,q')$ then
13:        if $i < |w|$ and $w[i] = \sigma$ then $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q',i+1)$
14:        else $\mathsf{result} \leftarrow \mathsf{Failure}(\nu_{\mathcal{A}_0}(q))$
15:    else if $T(q) = \mathsf{Sub}(\mathsf{at},\mathcal{A}',q')$ then
16:        $(P',Q',{q_0}',F',T') = \mathcal{A}'$
17:        $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A}',w}(q_0',i)$
18:        if $\mathsf{result} = \mathsf{SuccessAt}(i',K')$ then
19:            $\mathsf{result} \leftarrow \textsc{Memo-at}^{M}_{\mathcal{A}_0,\mathcal{A},w}(q',i')$
20:            if $\mathsf{result} = \mathsf{SuccessAt}(i'',K'')$ then $\mathsf{result} \leftarrow \mathsf{SuccessAt}(i'',K' \cup K'')$
21:            else if $\mathsf{result} = \mathsf{Failure}(j)$ then
22:                for $k \in K'$ do $M(k) \leftarrow \mathsf{Failure}(j)$
23:        else if $\mathsf{result} = \mathsf{Failure}(j)$ and $j > \nu_{\mathcal{A}_0}(q)$ then $\mathsf{result} \leftarrow \mathsf{Failure}(\nu_{\mathcal{A}_0}(q))$
24:    if $\mathsf{result} = \mathsf{SuccessAt}(i',K)$ then $\mathsf{result} \leftarrow \mathsf{SuccessAt}(i',K \cup \{(q,i)\})$
25:    else if $\mathsf{result} = \mathsf{Failure}(j)$ then $M(q,i) \leftarrow \mathsf{Failure}(j)$
26:    return $\mathsf{result}$
```

We also define the maximum nesting depth of atomic grouping for $\mathcal{A}$*, denoted by* $\nu(\mathcal{A})$*, as* $\nu(\mathcal{A}) = \max_{q \in P} \nu_{\mathcal{A}}(q)$.

Algorithm 7 is our algorithm for at-NFAs; the type of its memoization tables is $M\colon P \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}(j) \mid j \in \{0,\dots,\nu(\mathcal{A})\}\}$. Some remarks are in order.

Note first that the algorithm takes, as its parameters, the whole at-NFA $\mathcal{A}_0$ and its sub-automaton $\mathcal{A}$ as the algorithm's current scope. The top-level call is made with $\mathcal{A}_0 = \mathcal{A}$ (cf. Thm. 5 and 6); when an at transition is encountered, the scope goes to the corresponding sub-automaton ($\mathcal{A}'$ in Line 17).

In Line 9, the **if** condition checks that the nesting depth of $\mathsf{Failure}$ is the depth of the current NFA, and backtracking is performed if and only if it is true. This approach is crucial for avoiding the error in Example 4. The rest of the cases for Eps, Branch, Char is similar to Algorithm 5.

The case for Sub (Lines 15–23) requires some explanation. It is an adaptation of Lines 17–21 of Algorithm 5 with memoization. The apparent complication comes from the set K in $\mathsf{SuccessAt}(i', K)$. The set K is a set of *keys* for a memoization table M, that is, pairs (q, i) of a state and a position. The role of K is to collect the set of keys of M for which, once failure happens, the entry $\mathsf{Failure}(j)$ has to be recorded (this is done in a batch manner in Line 22). More specifically, once failure happens in an outer automaton (i.e., at a smaller depth j), this has to be recorded as $\mathsf{Failure}(j)$ for inner automata (at greater depths). The set K collects those keys for which this has to be done, starting from inner automata ($\mathcal{A}'$, Line 18) and going to outer ones ($\mathcal{A}$, Lines 19–20).

A closer inspection reveals that Line 20 is vacuous in Algorithm 7; however, it is needed when we combine it with look-around at the end of the section.

Algorithm 7 exhibits the desired properties, namely correctness (with respect to Prob. 2) and linear-time complexity. In Thm. 6, f is a function that converts results of Algorithm 7 to results of Algorithm 2; it is defined by $f(\mathsf{Failure}(j)) = \mathsf{Failure}$ and $f(\mathsf{SuccessAt}(i', K)) = \mathsf{SuccessAt}(i')$.

Theorem 5 (linear-time complexity of Algorithm 7). *For an at-NFA $\mathcal{A} = (P, Q, q_0, F, T)$, an input string w, and an position $i \in \{0, \dots, |w|\}$, $\textsc{Memo}\text{-at}^{M_0}_{\mathcal{A},\mathcal{A},w}(q_0, i)$ terminates with $O(|w|)$ recursive calls.*

Theorem 6 (correctness of Algorithm 7). *For an at-NFA $\mathcal{A} = (P, Q, q_0, F, T)$, an input string w, and an position $i \in \{0, \dots, |w|\}$, $\textsc{Match}\text{-(la, at)}(q_0, i) = f(\textsc{Memo}\text{-at}^{M_0}_{\mathcal{A},\mathcal{A},w}(q_0, i))$.*

Thm. 5 and 6 are proved similarly to Thm. 1 and 2; see [15, Appendix B.3]. The following points require some extra care.

Firstly, for linear-time complexity (Thm. 5), there is another recursive call (Line 19) before the return value of a recursive call (Line 17) is memoized (Line 22). If the second recursive call (Line 19) eventually leads to (the same call as) the first call (Line 17) (let's call this event $(*)$), then this can nullify the effect of memoization. We prove, as a lemma, that $(*)$ never happens.

Secondly, for correctness (Thm. 6), our conversion of runs should replace an invocation of the memoization table—if it returns a failure with a shallower depth—with not only the corresponding run (as before) but also the run of the second recursive call (Line 19). See [15, Appendix B.3] for details.

Combination with Look-around It is also possible to combine with Algorithm 6 (for look-around) and Algorithm 7 (for atomic grouping). In this case, the type of memoization tables becomes $M \colon P \times \mathbb{N} \rightharpoonup \{\mathsf{Failure}(j) \mid j \in \{0, \dots, \nu(\mathcal{A})\}\} \cup \{\mathsf{Success}\}$ and nesting depths of the atomic group are reset by look-around operators. A complete algorithm can be found in [15, Appendix C]; it also exhibits the desired properties.

6 Experiments and Evaluation

Implementation We implemented the algorithm proposed in this paper for evaluation. We call our implementation `memo-regex`. It is written in 1368 lines of Scala.

Table 2: our benchmark regexes and input strings

r_1	`/^(?=^.{1,254}$)(^(?:(?!\.\|-)([a-z0-9\-\*]{1,63}\|([a-z0-9\-]{1,62}[a-z0-9]))\.)+(?:[a-z]{2,})$)$/s`
	input: $w_1 =$ "0." "0.0a."^{n} "\\$u$0000", complexity: $O(2^n)$
	`https://regexlib.com/REDetails.aspx?regexp_id=3494`

r_2	`/(?=(?:[^\'']*\'[^\'']*\')*(?![^\'']*\'))/`
	input: $w_2 =$ "x"^{n} "$'$", complexity: $O(n^2)$
	`https://regexlib.com/REDetails.aspx?regexp_id=938`

r_3	`/(?<=[\w\s](?:[\.\!\? ]+[\x20]*[\x22\xBB]*))(?:\s+(?![\x22\xBB](?!\w)))/`
	input: $w_3 =$ "\\”"^{n} " ", complexity: $O(n^2)$
	`https://regexlib.com/REDetails.aspx?regexp_id=2355`

r_4	`/(?:(<)\s*?(\w+)(\s*?(?>(?!=[\/\?]?>)(\w+)(?:\s*(=)\s*)((?:\'[^\'']*\'|\"[^\"]*\"|[^ >]+))))\s*?([\/\?]?>))/`
	input: $w_4 =$ "<" "aaa"^{n} ">", complexity: $O(n^2)$
	`https://regexlib.com/REDetails.aspx?regexp_id=373`

memo-regex supports both look-around (i.e., look-ahead and look-behind) and atomic grouping. We implemented a regex parser ourselves. Backtracking is implemented by managing a stack manually rather than using a recursive function to prevent stack overflow. In this case, the memoization keys are pushed onto the stack. Recoding these keys in a memoization table is done during backtracking. We used the mutable `HashMap` from the Scala standard library as a data structure for memoization tables.

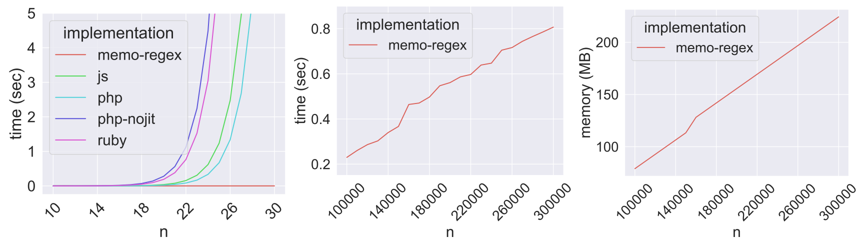
memo-regex also supports capturing sub-matchings. However, this feature cannot be used within atomic grouping and positive look-around because submatching information is lost for memoization.

The code of memo-regex, as well as all experiment scripts, is available [14].

Efficiency of Our Algorithm We conducted experiments to assess the performance of our memo-regex, in particular in comparison with other existing implementations.

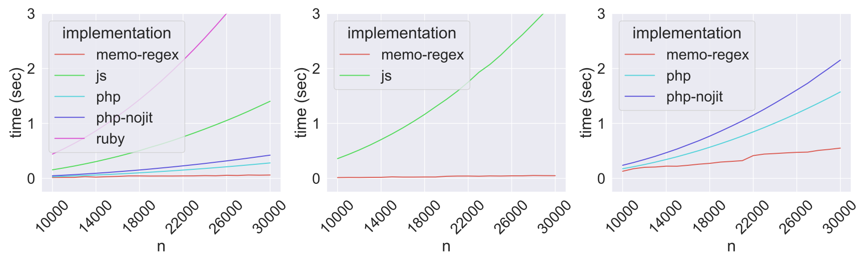
As target regexes, we looked for those with look-around and/or atomic grouping in the real-world regexes posted on `regexlib.com`. We then identified—by manual inspection—four regexes $r_1, \ldots, r_4$ that are subject to potential catastrophic backtracking. These regexes are shown in Table 2. We then crafted input strings $w_1, \ldots, w_4$, respectively, so that they cause catastrophic backtracking. Specifically, r_1 contains positive look-ahead and negative look-ahead; this positive look-ahead is used for restricting the length of input strings. The regexes r_2 and r_3 are themselves positive look-ahead and look-behind, respectively; both include negative look-ahead, too. The regex r_4 includes atomic grouping and negative look-ahead.

For these regexes, we measured matching time using memo-regex on OpenJDK 21.0.1. We compared it with the following implementations: Node.js 20.5.0,



(a) comparison to others (b) `memo-regex` only (c) memoization table size

Fig. 7: result for r_1



(a) r_2 (b) r_3 (c) r_4

Fig. 8: matching time for r_2, r_3 and r_4

Ruby 3.1.4, and PCRE2 10.42 (used by PHP 8.3.1, w/ or w/o JIT). All of these implementations use backtracking; Ruby and PCRE2 have restrictions on regexes inside look-behind and Node.js does not support atomic grouping. The experiments were performed 10 times and the average was adopted. Furthermore, for `memo-regex`, we measured the size of its memoization table by the memory usage, using jamm.[8] The experiments were conducted on MacBook Pro 2021 (Apple M1 Pro, RAM: 32 GB).

We show the results in Figures 7 and 8. Note that the values of n are different depending on whether the matching time complexity is $O(n^2)$ or $O(2^n)$. Results for some implementations are absent for r_3 and r_4 because of the syntactic restrictions discussed above.

In Figures 7 and 8, we observe clear performance advantages of `memo-regex`. In particular, its linear-time complexity and linear memory consumption (memoization table size) are experimentally confirmed.

Real-world Usage of Look-around and Atomic Grouping We additionally surveyed the use of the regex extensions of our interest, in order to confirm their practical relevance.

We used a regex dataset collected by a 2019 survey [9]. This dataset contains 537,806 regexes collected from the source code of real-world products.

[8] `https://github.com/jbellis/jamm`

We tallied the usage of each regex extension by parsing these regexes in the dataset with our parser in `memo-regex`. 8,679 regexes could not be parsed or compiled; this is due to back-reference for 4,360 regexes, unsupported syntax (Unicode character class, conditional branching, etc.) for 4,134 regexes, and too large or semantically invalid regexes for the other 184 regexes. We adopted the remaining 529,127 regexes for tallying.

The result is shown in Table 3. Note that 1) the numbers for look-ahead and look-behind do not include simple zero-width assertions such as `^` (line-begin) or `$` (line-end), and 2) that of atomic grouping includes possessive quantifiers such as `*+` and `++`.

In Table 3, we observe that 17,167 regexes (3.2%) in the dataset use at least one of the extensions we studied in this paper. While the ratio is not very large, the absolute number (17,167 regexes) is significant; this implies that there are a number of applications (such as web services) that rely on the regex extensions. Thereby we confirm the practical relevance of these regex extensions.

Table 3: regex ext. usage

feature	# of regexes
(total)	529,127
positive look-ahead	7,476 (1.4%)
negative look-ahead	6,917 (1.3%)
positive look-behind	1,746 (0.3%)
negative look-behind	3,750 (0.7%)
atomic grouping	1,113 (0.2%)
at least one of the above	17,167 (3.2%)

7 Conclusions and Future Work

In this paper, we proposed a backtracking algorithm with memoization for regexes with look-around and atomic grouping. It is the first linear-time backtracking matching algorithm for such regexes. It also fixs problems of the memoization matching algorithm in [10] for look-ahead. We implemented the algorithm; our experimental evaluation confirms its performance advantage.

One direction of future work is to support more extensions. Our implementation does not support a widely used regex extension, namely back-references. In the recent work [10], back-reference was supported by additionally recording captured positions in memoization tables. We expect that a similar idea is applicable to our algorithm.

Combination with selective memoization (used in [10]; see [15, Appendix A]) is another direction. We believe it is possible, but it will require a more detailed discussion on how to handle sub-automata in the selective memoization schema.

Acknowledgments

Thanks are due to Konstantinos Mamouras for pointing to [22] after the dissemination of the preprint version of this paper.

Data-Availability Statement

The data that support the findings of this study are openly available in Zenodo at `10.5281/zenodo.10458317`, reference number [14].

References

1. Aho, A.V.: Algorithms for finding patterns in strings. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, pp. 255–300. Elsevier and MIT Press (1990)
2. Berglund, M., Drewes, F., van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. In: Ésik, Z., Fülöp, Z. (eds.) Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014. EPTCS, vol. 151, pp. 109–123 (2014). https://doi.org/10.4204/EPTCS.151.7, `https://doi.org/10.4204/EPTCS.151.7`
3. Berglund, M., van der Merwe, B., van Litsenborgh, S.: Regular expressions with lookahead. J. Univers. Comput. Sci. **27**(4), 324–340 (2021). https://doi.org/10.3897/jucs.66330, `https://doi.org/10.3897/jucs.66330`
4. Berglund, M., van der Merwe, B., Watson, B.W., Weideman, N.: On the semantics of atomic subgroups in practical regular expressions. In: Carayol, A., Nicaud, C. (eds.) Implementation and Application of Automata - 22nd International Conference, CIAA 2017, Marne-la-Vallée, France, June 27-30, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10329, pp. 14–26. Springer (2017). https://doi.org/10.1007/978-3-319-60134-2"˙2, `https://doi.org/10.1007/978-3-319-60134-2\_2`
5. Chida, N., Terauchi, T.: On lookaheads in regular expressions with backreferences. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 228, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.15, `https://doi.org/10.4230/LIPIcs.FSCD.2022.15`
6. Chida, N., Terauchi, T.: Repairing dos vulnerability of real-world regexes. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022. pp. 2060–2077. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833597, `https://doi.org/10.1109/SP46214.2022.9833597`
7. Cox, R.: Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...) (2007), URL: `http://swtch.com/rsc/regexp/regexp1.html`
8. Cox, R.: Regular expression matching: the virtual machine approach (2009), URL: `http://swtch.com/~rsc/regexp/regexp2.html`
9. Davis, J.C., IV, L.G.M., Coghlan, C.A., Servant, F., Lee, D.: Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. pp. 443–454. ACM (2019). https://doi.org/10.1145/3338906.3338909, `https://doi.org/10.1145/3338906.3338909`
10. Davis, J.C., Servant, F., Lee, D.: Using selective memoization to defeat regular expression denial of service (redos). In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1–17. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00032, `https://doi.org/10.1109/SP40001.2021.00032`

11. Davis, J.: PurdueDualityLab/memoized-regex-engine: IEEE S&P21́ artifact (Apr 2021). https://doi.org/10.5281/zenodo.4718966, URL: https://zenodo.org/record/4718966
12. Exchange, S.: Stack exchange network status — outage postmortem - july 20, 2016. URL: http://web.archive.org/web/20210308040819/https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016 (2016)
13. Friedl, J.E.F.: Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.). O'Reilly (2006), http://www.oreilly.de/catalog/regex3/index.html
14. Fujinami, H., Hasuo, I.: Artifact Archive for memo-regex: linear-time regex matching implementation with memoization (Jan 2024). https://doi.org/10.5281/zenodo.10458317
15. Fujinami, H., Hasuo, I.: Efficient matching with memoization for regexes with look-around and atomic grouping (extended version) (2024), http://arxiv.org/abs/2401.12639
16. Graham-Cumming, J.: Details of the cloudflare outage on july 2, 2019 (2019), URL: https://web.archive.org/web/20190712160002/https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/
17. Herczeg, Z.: Extending the PCRE library with static backtracking based just-in-time compilation support. In: Kaeli, D.R., Moseley, T. (eds.) 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014. p. 306. ACM (2014), https://dl.acm.org/citation.cfm?id=2544146
18. Kirrage, J., Rathnayake, A., Thielecke, H.: Static analysis for regular expression denial-of-service attacks. In: López, J., Huang, X., Sandhu, R.S. (eds.) Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7873, pp. 135–148. Springer (2013). https://doi.org/10.1007/978-3-642-38631-2"˙11, https://doi.org/10.1007/978-3-642-38631-2_11
19. Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L., Cheung, S.: ReDoSHunter: A combined static and dynamic approach for regular expression dos detection. In: Bailey, M., Greenstadt, R. (eds.) 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021. pp. 3847–3864. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/li-yeting
20. Li, Y., Sun, Y., Xu, Z., Cao, J., Li, Y., Li, R., Chen, H., Cheung, S., Liu, Y., Xiao, Y.: RegexScalpel: Regular expression denial of service (redos) defense by localize-and-fix. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 4183–4200. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting
21. Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S., Zhao, H.: FlashRegex: Deducing anti-redos regexes from examples. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. pp. 659–671. IEEE (2020). https://doi.org/10.1145/3324884.3416556, https://doi.org/10.1145/3324884.3416556
22. Mamouras, K., Chattopadhyay, A.: Efficient matching of regular expressions with lookaround assertions. Proc. ACM Program. Lang. **8**(POPL), 2761–2791 (2024). https://doi.org/10.1145/3632934, https://doi.org/10.1145/3632934

23. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Trans. Electron. Comput. **9**(1), 39–47 (1960). https://doi.org/10.1109/TEC.1960.5221603, `https://doi.org/10.1109/TEC.1960.5221603`
24. van der Merwe, B., Mouton, J., van Litsenborgh, S., Berglund, M.: Memoized regular expressions. In: Maneth, S. (ed.) Implementation and Application of Automata - 25th International Conference, CIAA 2021, Virtual Event, July 19-22, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12803, pp. 39–52. Springer (2021). https://doi.org/10.1007/978-3-030-79121-6"˙4, `https://doi.org/10.1007/978-3-030-79121-6\_4`
25. Michael, IV, L.G., Donohue, J., Davis, J.C., Lee, D., Servant, F.: Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. pp. 415–426. IEEE (2019). https://doi.org/10.1109/ASE.2019.00047, `https://doi.org/10.1109/ASE.2019.00047`
26. Miyazaki, T., Minamide, Y.: Derivatives of regular expressions with lookahead. J. Inf. Process. **27**, 422–430 (2019). https://doi.org/10.2197/ipsjjip.27.422, `https://doi.org/10.2197/ipsjjip.27.422`
27. Morihata, A.: Translation of regular expression with lookahead into finite state automaton. Computer Software **29**(1), 1_147–1_158 (2012). https://doi.org/10.11309/jssst.29.1˙147
28. Moseley, D., Nishio, M., Rodriguez, J.P., Saarikivi, O., Toub, S., Veanes, M., Wan, T., Xu, E.: Derivative based nonbacktracking real-world regex matching with backtracking semantics. Proc. ACM Program. Lang. **7**(PLDI), 1026–1049 (2023). https://doi.org/10.1145/3591262, `https://doi.org/10.1145/3591262`
29. Rathnayake, A., Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. CoRR **abs/1405.7058** (2014), `http://arxiv.org/abs/1405.7058`
30. Sakuma, Y., Minamide, Y., Voronkov, A.: Translating regular expression matching into transducers. J. Appl. Log. **10**(1), 32–51 (2012). https://doi.org/10.1016/j.jal.2011.11.003, `https://doi.org/10.1016/j.jal.2011.11.003`
31. Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., Lu, J.: ReScue: crafting regular expression dos attacks. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 225–235. ACM (2018). https://doi.org/10.1145/3238147.3238159, `https://doi.org/10.1145/3238147.3238159`
32. Sidhu, R.P.S., Prasanna, V.K.: Fast regular expression matching using fpgas. In: The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2001, Rohnert Park, California, USA, April 29 - May 2, 2001. pp. 227–238. IEEE Computer Society (2001). https://doi.org/10.1109/FCCM.2001.22, `https://doi.ieeecomputersociety.org/10.1109/FCCM.2001.22`
33. Staicu, C., Pradel, M.: Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 361–376. USENIX Association (2018), `https://www.usenix.org/conference/usenixsecurity18/presentation/staicu`

34. Sugiyama, S., Minamide, Y.: Checking time linearity of regular expression matching based on backtracking. Information and Media Technologies **9**(3), 222–232 (2014). https://doi.org/10.11185/imt.9.222
35. Thompson, K.: Regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968). https://doi.org/10.1145/363347.363387, `https://doi.org/10.1145/363347.363387`
36. Weideman, N., van der Merwe, B., Berglund, M., Watson, B.W.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: Han, Y., Salomaa, K. (eds.) Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9705, pp. 322–334. Springer (2016). https://doi.org/10.1007/978-3-319-40946-7“˙27, `https://doi.org/10.1007/978-3-319-40946-7\_27`
37. Wüstholz, V., Olivo, O., Heule, M.J.H., Dillig, I.: Static detection of dos vulnerabilities in programs that use regular expressions. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10206, pp. 3–20 (2017). https://doi.org/10.1007/978-3-662-54580-5“˙1, `https://doi.org/10.1007/978-3-662-54580-5\_1`

Verification

A Denotational Approach to Release/Acquire Concurrency

Yotam Dvir[1](✉), Ohad Kammar[2], and Ori Lahav[1]

[1] Tel Aviv University, Tel Aviv, Israel
yotamdvir@mail.tau.ac.il, orilahav@tau.ac.il
[2] University of Edinburgh, Edinburgh, UK
ohad.kammar@ed.ac.uk

Abstract. We present a compositional denotational semantics for a functional language with first-class parallel composition and shared-memory operations whose operational semantics follows the Release/Acquire weak memory model (RA). The semantics is formulated in Moggi's monadic approach, and is based on Brookes-style traces. To do so we adapt Brookes's traces to Kang et al.'s view-based machine for RA, and supplement Brookes's mumble and stutter closure operations with additional operations, specific to RA. The latter provides a more nuanced understanding of traces that uncouples them from operational interrupted executions. We show that our denotational semantics is adequate and use it to validate various program transformations of interest. This is the first work to put weak memory models on the same footing as many other programming effects in Moggi's standard monadic approach.

Keywords: Weak memory models · Release/Acquire · Shared state · Shared memory · Concurrency · Denotational semantics · Monads · Program refinement · Program equivalence · Compiler optimizations

1 Introduction

Denotational semantics defines the meaning of programs *compositionally*, where the meaning of a program term is a function of the meanings assigned to its immediate syntactic constituents. This key feature makes denotational semantics instrumental in understanding the meaning a piece of code independently of the context under which the code will run. This style of semantics contrasts with standard operational semantics, which only executes closed/whole programs. A basic requirement of such a denotation function $[\![-]\!]$ is for it to be *adequate* w.r.t. a given operational semantics: plugging program terms M and N with equal denotations—i.e. $[\![M]\!] = [\![N]\!]$—into some program context $\Xi[-]$ that closes over their variables, results in observationally indistinguishable closed programs in the given operational semantics. Moreover, assuming that denotations have a defined order ($\leq$), a "directed" version of adequacy ensures that $[\![M]\!] \leq [\![N]\!]$ implies that all behaviors exhibited by $\Xi[M]$ under the operational semantics are also exhibited by $\Xi[N]$.

For shared-memory concurrent programming, Brookes's seminal work [13] defined a denotational semantics, where the denotation $[\![M]\!]$ is a set of totally ordered traces of M closed under certain operations, called stutter and mumble. Traces consist of sequences of memory snapshots that M guarantees to provide while relying on its environment to make other memory snapshots. Brookes [12] used the insights behind this semantics to develop a semantic model for separation logic, and Turon and Wand [46] used them to design a separation logic for refinement. Additionally, Xu et al. [48] used traces as a foundation for the Rely/Guarantee approach for verification of concurrent programs, and Liang et al., Liang et al. [34, 35] used a trace-based program logic for refinement.

A *memory model* decides what outcomes are possible from the execution of a program. Brookes established the adequacy of the trace-based denotational semantics w.r.t. the operational semantics of the strongest model, known as *sequential consistency* (SC), where every memory access happens instantaneously and immediately affects all concurrent threads. However, SC is too strong to model real-world shared memory, whether it be of modern hardware, such as x86-TSO [40, 44] and ARM, or of programming languages such as C/C++ and Java [4, 37]. These runtimes follow *weak memory models* that allow performant implementations, but admit more behaviors than SC.

Do weak memory models admit adequate Brookes-style denotational semantics? This question has been answered affirmatively once, by Jagadeesan et al. [25], who closely followed Brookes to define denotational semantics for x86-TSO. Other weak memory models, in particular, models of *programming languages*, and *non-multi-copy-atomic* models, where writes can be observed by different threads in different orders, have so far been out of reach of Brookes's totally ordered traces, and were only captured by much more sophisticated models based on *partial orders* [15, 19, 24, 26, 28, 41].

In this paper we target the Release/Acquire memory model (RA, for short). This model, obtained by restricting the C/C++11 memory model to Release/ Acquire atomics, is a well-studied fundamental memory model weaker than x86-TSO, which, roughly speaking, ensures "causal consistency" together with "per-location-SC" and "RMW (read-modify-write) atomicity" [29, 30]. These assurances make RA sufficiently strong for implementing common synchronization idioms. RA allows more performant implementations than SC, since, in particular, it allows the reordering of a write followed by a read from a different location, which is commonly performed by hardware, and it is non-multi-copy-atomic, thus allowing less centralized architectures like POWER [45].

Our first contribution is a Brookes-style denotational semantics for RA. As Brookes's traces are totally ordered, this result may seem counterintuitive. The standard semantics for RA is a declarative (a.k.a. axiomatic) memory model, in the form of acyclicity consistency constraints over partially ordered candidate execution graphs. Since these graphs are not totally ordered, one might expect that Brookes's traces are insufficient. Nevertheless, our first key observation is that an *operational* presentation of RA as an interleaving semantics of a weak memory system lends itself to Brookes-style semantics. For that matter, we develop a notion of traces compatible with Kang et al.'s "view-based" machine [27], an operational semantics that is equivalent to RA's declarative formulation. Our main technical result is the (directed) adequacy of the proposed Brookes-style semantics w.r.t. that operational semantics of RA.

A main challenge when developing a denotational semantics lies in making it sufficiently abstract. While *full* abstraction is often out of reach, as a yardstick, we want our semantics to be able to justify various compiler transformations/optimizations that are known to be sound under RA [47]. Indeed, an immediate practical application of a denotational semantics is the ability to provide *local* formal justifications of program transformations, such as those performed by optimizing compilers. In this setting, to show that an optimization $N \twoheadrightarrow M$ is valid amounts to showing that replacing N by M anywhere in a larger program does not introduce new behaviors, which follows from $[\![M]\!] \leq [\![N]\!]$ given a directionally adequate denotation function $[\![-]\!]$.

To support various compiler transformations, we close our denotations under certain operations, including analogs to Brookes's stutter and mumble, but also several RA-specific operations, that allow us to relate programs which would naively correspond to rather different sets of traces. Given these closure operations, our semantics validates standard program transformations, including structural transformations, algebraic laws of parallel programming, and all known thread-local RA-valid compiler optimizations. Thus, the denotational semantics is instrumental in formally establishing validity of transformations under RA, which is a non-trivial task [19, 47].

Our second contribution is to connect the core semantics of parallel programming languages exhibiting weak behaviors to the more standard semantic account for programming languages with effects. Brookes presented his semantics for a simple imperative WHILE language, but Benton et al., Dvir et al. [6, 20] later recast it atop Moggi's monad-based approach [38] which uses a functional, higher-order core language. In this approach the core language is modularly extended with effect constructs to denote program effects. In particular, we define parallel composition as a first-class operator. This is in contrast to most of the research of weak memory models that employ imperative languages and assume a single top-level parallel composition.

A denotational semantics given in this monadic style comes ready-made with a rich semantic toolkit for program denotation [7], transformations [5, 8–10, 23], reasoning [2, 36], etc.. We challenge and reuse this diverse toolkit throughout the development. We follow a standard approach and develop specialized logical relations to establish the compositionality property of our proposed semantics; its soundness, which allows one to use the denotational semantics to show that certain outcomes are impossible under RA; and adequacy. This development puts weak memory models, which often require bespoke and highly specialized presentations, on a similar footing to many other programming effects.

Outline. In §2 we lay the groundwork for the rest of the paper by introducing the programming language that we will use (§2.1), the main ideas that underpin Brookes's trace-based denotational semantics (§2.2), and the operational RA model (§2.3). In §3 we present the core aspects of our denotational semantics. First, we discuss our extension of RA's operations semantics with first-class parallelism, which enables denotations to be defined for concurrent composition (§3.1). We then present RA traces (§3.2) and use them to define the denotations of key program constructs (§3.3). Next, we show how the restriction of traces within denotations (§3.4) and the addition of closure operations (§3.5) make our denotational semantics more abstract. The denotational semantics extends to the entire programming language standardly using Moggi's monad-based approach (§3.6). With the denotational semantics in place, we present our main results in §4. Finally, we conclude and discuss related work in §5. More details are available in the extended version of this paper [21].

2 Preliminaries

We first introduce the language and its operational semantics under the Sequential Consistency (SC) memory model (§2.1). We then outline Brookes's denotational semantics for SC (§2.2). Finally, we introduce Kang et al.'s operational presentation of Release/Acquire (RA) (§2.3).

2.1 Language and Operational Semantics

The programming language we use is an extension of a functional language with shared-state constructs. Program terms M and N can be composed sequentially explicitly as $M\,;N$ or implicitly by left-to-right evaluation in the pairing construct $\langle M, N\rangle$. They can be composed in parallel as $M \parallel N$. We assume preemptive scheduling, thus imposing no restrictions on the interleaving execution steps between parallel threads. To introduce the memory-access constructs, we present the well-known *message passing* litmus test, adapted to the functional setting:

$$(\mathtt{x} := 1\,;\mathtt{y} := 1) \parallel \langle \mathtt{y}?, \mathtt{x}? \rangle \qquad \text{(MP)}$$

Here, x and y refer to distinct shared memory locations. Assignment $\ell := v$ stores the value v at location ℓ in memory, and dereference $\ell?$ loads a value from ℓ. The language also includes atomic read-modify-write (RMW) constructs. For example, assuming integer storable values, FAA (ℓ, v) (Fetch-And-Add) atomically adds v to the value stored in ℓ. In contrast, interleaving is permitted between the dereferencing, adding, and storing in $\ell := (\ell? + v)$. The underlying *memory model* dictates the behavior of the memory-access constructs more specifically.

In the functional setting, execution results in a returned value: $\ell := v$ returns the unit value $\langle\rangle$, i.e. the empty tuple; $\ell?$, and the RMW constructs such as FAA (ℓ, v), return the loaded value; $M\,;N$ returns what N returns; and $\langle M, N\rangle$, as well as $M \parallel N$, return the pair consisting of the return value of M and the return value of N. We assume left-to-right execution of pairs, so in the (MP) example $\langle \mathtt{y}?, \mathtt{x}?\rangle$ steps to $\langle v, \mathtt{x}?\rangle$ for a value v that can be loaded from y, and $\langle v, \mathtt{x}?\rangle$ steps to $\langle v, w\rangle$ for a value w that can be loaded from x. In between, the left side of the parallel composition ($\parallel$) can take steps.

We can use intermediate results in subsequent computations via let binding: $\mathbf{let}\, a = M\, \mathbf{in}\, N$ binds the result of M to a in N. Thus, we execute M first, and substitute the resulting value V for a in N before executing $N[a \mapsto V]$. Similarly, we deconstruct pairs by matching: $\textbf{match}\, M \,\textbf{with}\, \langle a, b\rangle .\, N$ binds the components of the pair that M returns to a and b respectively in N. The first and second projections **fst** and **snd**, as well as the operation **swap** that swaps the pair constituents, are defined using **match** standardly.

Sequential consistency. In the strongest memory model of Sequential Consistency (SC), every value stored is immediately made available to every thread, and every dereference must load the latest stored value. Thus the underlying memory model uses maps from locations to values for the memory state that evolves during program execution. Given an initial state, the behavior of a program in SC depends only on the choice of interleaving of steps. Though any such map can serve as an initial state, litmus tests are traditionally designed with the memory that sets all values to 0 in mind. In (MP) the order of the two stores and the two loads ensures that executions under SC may return $\langle\langle\rangle, \langle 0, 0\rangle\rangle$, $\langle\langle\rangle, \langle 0, 1\rangle\rangle$, and $\langle\langle\rangle, \langle 1, 1\rangle\rangle$, *but not* $\langle\langle\rangle, \langle 1, 0\rangle\rangle$.

Observations. An *observable behavior* of an entire program is a value it may evaluate to from given initial memory values. While programs may internally interact and observe the memory, we do not consider it feasible to observe the memory directly.

2.2 Overview of Brookes's Trace-based Semantics

Observable behavior as defined for whole programs is too crude to study program *terms* that can interact with the program context within which they run. Indeed, compare M_1 defined as $\mathtt{x} := 1\,;\mathtt{y} := \boxed{1}\,;\mathtt{y?}$ versus M_2 defined as $\mathtt{x} := 1\,;\mathtt{y} := \boxed{\mathtt{x?}}\,;\mathtt{y?}$. Under SC, the difference between them as whole programs is unobservable: starting from any initial state both return 1. Now consider them within the program context $- \parallel \mathtt{x} := 2$. That is, compare $M_1 \parallel \mathtt{x} := 2$ versus $M_2 \parallel \mathtt{x} := 2$. In the first, M_1 still always returns 1; but in the second, M_2 can also return 2 by interleaving the store of 2 in x immediately after the store of 1 in x. Thus, if $[\![M]\!]$, i.e. M's denotation, were to simply map initial states to possible results according to executions of M, we could not define $[\![M \parallel N]\!]$ in terms of $[\![M]\!]$ and $[\![N]\!]$ alone, because we would have $[\![M_1]\!] = [\![M_2]\!]$ but also $[\![M_1 \parallel \mathtt{x} := 2]\!] \neq [\![M_2 \parallel \mathtt{x} := 2]\!]$. We conclude that $[\![M]\!]$ must contain more information on M than an "input-output" relation; it must account for interference by the environment.

Adequacy in SC. A prominent approach to define compositional semantics for concurrent programs is due to Brookes [13], who defined a denotational semantics for SC by taking $[\![M]\!]$ to be a set of traces of M closed under certain rewrite rules as we detail below. Brookes established a (directional) adequacy theorem: if $[\![M]\!] \supseteq [\![N]\!]$ then the transformation $M \twoheadrightarrow N$ is valid under SC. The latter means that, when assuming SC-based operational semantics, M can be replaced by N within a program without introducing new observable behaviors for it.

Thus, adequacy formally grounds the intuition that the denotational semantics soundly captures behavior of program terms.

As a particular practical benefit, formal and informal simulation arguments which are used to justify transformations in operational semantics can be replaced by cleaner and simpler proofs based on the denotational semantics. For example, a simple argument shows that $[\![\mathtt{x} := v \,;\, \mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ holds in Brookes's semantics. Thanks to adequacy, this justifies Write-Write Elimination (WW-Elim) $\mathtt{x} := v \,;\, \mathtt{x} := w \twoheadrightarrow \mathtt{x} := w$ in SC.

Traces in SC. In Brookes's semantics, a program term is denoted by the set of traces, each trace consisting of a sequence of transitions. Each transition is of the form $\langle \mu, \rho \rangle$, where μ and ρ are memories, i.e. maps from locations to values. A transition describes a program term's execution relying on a memory state μ in order to guarantee the memory state ρ.

For example, $[\![\mathtt{x} := w]\!]$ includes all traces of the form $\boxed{\langle \rho, \rho\,[\mathtt{x} := w] \rangle}$, where $\rho\,[\mathtt{x} := w]$ is equal to ρ except for mapping $\mathtt{x}$ to w. The definition is compositional: the traces in $[\![\mathtt{x} := v \,;\, \mathtt{x} := w]\!]$ are obtained from sequential compositions of traces from $[\![\mathtt{x} := v]\!]$ with traces from $[\![\mathtt{x} := w]\!]$, obtaining all traces of the form $\boxed{\langle \mu, \mu\,[\mathtt{x} := v] \rangle \, \langle \rho, \rho\,[\mathtt{x} := w] \rangle}$. Such a trace relies on μ in order to guarantee $\mu\,[\mathtt{x} := v]$, and then relies on ρ in order to guarantee $\rho\,[\mathtt{x} := w]$. Allowing $\rho \neq \mu\,[\mathtt{x} := v]$ reflects the possibility of environment interference between the two store instructions. Indeed, when denoting parallel composition $[\![M \parallel N]\!]$ we include all traces obtained by interleaving transitions from a trace from $[\![M]\!]$ with transitions from a trace from $[\![N]\!]$. By sequencing and interleaving, one subterm's guarantee can fulfill the requirement which another subterm relies on. They may also relegate reliances and guarantees to their mutual context.

In the functional setting, executions not only modify the state but also return values. In this setting, traces are pairs, which we write as $\boxed{\xi} \mathrel{.\!\cdot\!.} r$, where ξ is the sequence of transitions and r represents the final value that the program term guarantees to return [6]. For example, the semantics of dereference $[\![\mathtt{x?}]\!]$ includes all traces of the form $\boxed{\langle \mu, \mu \rangle} \mathrel{.\!\cdot\!.} \mu(\mathtt{x})$. Indeed, the execution of $\mathtt{x?}$ does not change the memory and returns the value loaded from $\mathtt{x}$. In the semantics of assignment $[\![\mathtt{x} := v]\!]$, instead of $\boxed{\langle \mu, \mu\,[\mathtt{x} := v] \rangle}$ we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := v] \rangle} \mathrel{.\!\cdot\!.} \langle\rangle$.

Rewrite rules in SC. Were denotations in Brookes's semantics defined to *only* include the traces explicitly mentioned above, it would not be abstract enough to justify (WW-Elim), which eliminates redundant writes. Indeed, we only saw traces with two transitions in $[\![\mathtt{x} := v \,;\, \mathtt{x} := w]\!]$, but in $[\![\mathtt{x} := w]\!]$ we saw traces with one. The semantics would still be adequate, but it would lack abstraction. This is where Brookes's second main idea comes into play, making the denotations more abstract by closing them under two operations that rewrite traces:

Stutter adds a transition of the form $\langle \mu, \mu \rangle$ anywhere in the trace. Intuitively, a program term can always guarantee what it relies on.

Mumble combines a couple of subsequent transitions of the form $\langle \mu, \rho \rangle \, \langle \rho, \theta \rangle$ into a single transition $\langle \mu, \theta \rangle$ anywhere in the trace. Intuitively, a program term can always omit a guarantee to the environment, and rely on its own omitted guarantee instead of relying on the environment.

Denotations in Brookes's semantics are defined to be sets of traces *closed* under rewrite rules: applying a rewrite to a trace in the set results in a trace that is also in the set. For example, $[\![\mathtt{x} := w]\!]$ is the least closed set with all traces of the form $\boxed{\langle \rho, \rho\,[\mathtt{x} := w]\rangle} \mathrel{.\cdot.} \langle\rangle$, and $[\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!]$ is the least closed set with all sequential compositions of traces from $[\![\mathtt{x} := v]\!]$ with traces from $[\![\mathtt{x} := w]\!]$.

Closure under these rules makes traces in $[\![M]\!]$ correspond precisely to *interrupted executions* of M, which are executions of M in which the memory can arbitrarily change between steps of execution. Each transition $\langle \mu, \rho\rangle$ in a trace in $[\![M]\!]$ corresponds to multiple execution steps of M that transition μ into ρ, and each gap between transitions accounts for possible environment interruption. The rewrite rules maintain this correspondence: stutter corresponds to taking 0 steps, and mumble corresponds to taking $n + m$ steps instead of taking n steps and then m steps when the environment did not change the memory in between. Brookes's adequacy proof is based on this precise correspondence. In particular, the single-pair traces in $[\![M]\!]$ correspond to the (uninterrupted) executions, the “input-output” relation, of M.

Abstraction in SC. Brookes's semantics is *fully abstract*, meaning that the converse to adequacy also holds: if $N \twoheadrightarrow M$ is valid under SC, then $[\![N]\!] \supseteq [\![M]\!]$. However, Brookes's proof relies on an artificial program construct, **await**, that permits waiting for a specified memory snapshot and then step (atomically) to a second specified memory snapshot. Thus, in realistic languages, when this construct is unavailable, Brookes's full abstraction proof does not apply.

Nevertheless, even without full abstraction, one can still provide evidence that an adequate semantics is abstract by ensuring that it supports known transformations. As an example, we show directly that $[\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ holds in Brookes's semantics. Since $[\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!]$ is closed, it suffices to show that $[\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!] \supseteq \left\{ \boxed{\langle \mu, \mu\,[\mathtt{x} := w]\rangle} \mathrel{.\cdot.} \langle\rangle \;\middle|\; \text{memory } \mu \right\}$. For a memory μ, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := v]\rangle\, \langle \rho, \rho\,[\mathtt{x} := w]\rangle} \mathrel{.\cdot.} \langle\rangle \in [\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!]$ for every memory ρ, in particular when $\rho = \mu\,[\mathtt{x} := v]$. Since $\rho\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := v]\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := w]$, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := v]\rangle\, \langle \mu\,[\mathtt{x} := v], \mu\,[\mathtt{x} := w]\rangle} \mathrel{.\cdot.} \langle\rangle \in [\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!]$. After applying mumble, we have $\boxed{\langle \mu, \mu\,[\mathtt{x} := w]\rangle} \mathrel{.\cdot.} \langle\rangle \in [\![\mathtt{x} := v\,;\, \mathtt{x} := w]\!]$.

2.3 Overview of Release/Acquire Operational Semantics

Memory accesses in RA are more subtle in than in SC. To address this we adopt Kang et al.'s “view-based” machine [27], an operational presentation of RA proven to be equivalent to the original declarative formulation of RA [e.g. 30]. In this model, rather than the memory holding only the latest value written to every variable, the memory accumulates a set of memory update messages for each location. Each thread maintains its own *view* that captures which messages the thread can observe, and is used to constrain the messages that the thread

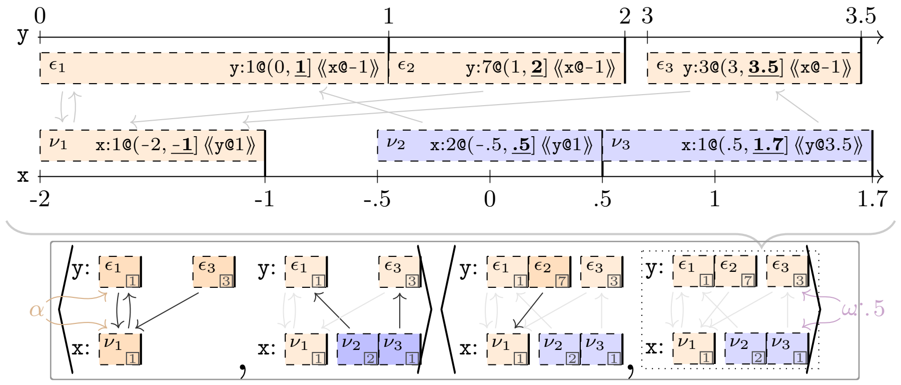

y
x
0
1
2
3
3.5
-2
-1
-.5
0
.5
1
1.7
ϵ_1 y:1@(0, 1] ⟨⟨x@-1⟩⟩
ϵ_2 y:7@(1, 2] ⟨⟨x@-1⟩⟩
ϵ_3 y:3@(3, 3.5] ⟨⟨x@-1⟩⟩
ν_1 x:1@(-2, -1] ⟨⟨y@1⟩⟩
ν_2 x:2@(-.5, .5] ⟨⟨y@1⟩⟩
ν_3 x:1@(.5, 1.7] ⟨⟨y@3.5⟩⟩
α
ω ∴ .5

Fig. 1. Illustrations of a memory (top) and a trace (bottom), in the setting of two memory locations, x and y. **Top:** A memory holding six messages. The timelines are purposefully misaligned and not to scale to emphasize that timestamps for different locations are incomparable and that only the order between them is relevant. The graph structure that the views impose is illustrated by arrows pointing between messages. Messages that are not dovetailed are set apart, e.g. ν_3 dovetails with ν_2, which does not dovetail with ν_1. **Bottom:** A trace with two transitions: $\alpha \boxed{\langle \mu_1, \rho_1 \rangle \, \langle \mu_2, \rho_2 \rangle} \omega \therefore 5$. The memory illustrated on top is ρ_2. Messages and edges that are not part of a previous memory are highlighted. The local messages are ν_2 and ν_3, and the rest are environment messages.

may read and write. The messages in the memory carry views as well, which are inherited from the thread that wrote the message, and passed to any thread that reads the message. Thus views indirectly maintain a causal relationship between messages in memory throughout the evolution of the system.

More concretely, causality is enforced by timestamping messages, thus placing them on their location's *timeline*. To capture the atomicity of RMWs, each message occupies a half-open segment $(q, t]$ on their location's timeline, where t is the message's timestamp. It *dovetails* with a message at the same location with timestamp q. An RMW "modifies" a message by dovetailing with it.

A view κ associates a timestamp $\kappa(\ell)$ to each location ℓ, obscuring the portion of ℓ's timeline before $\kappa(\ell)$. The view *points to* a message at ℓ with timestamp $\kappa(\ell)$. A view ω *dominates* a view α, written $\alpha \leq \omega$, if $\alpha(\ell) \leq \omega(\ell)$ for every ℓ.

Messages point to messages via the view they carry, and must point to themselves. So when specifying a message, the value its view takes at its location may be omitted. For example, assuming of two location, x and y, we denote by x:1@$(.5, \underline{\mathbf{1.7}}]$ ⟨⟨y@3.5⟩⟩ the message at location x that carries the value 1, occupies the segment $(.5, 1.7]$ on x's timeline, and carries the view κ such that $\kappa(\mathtt{x}) = 1.7$ and $\kappa(\mathtt{y}) = 3.5$. An example memory is depicted on the top of Figure 1.

When a thread writes to ℓ, it must increase the timestamp its view associates with ℓ and use its new view as the message's view. The message's segment must not overlap with any other segment on ℓ's timeline. In particular, only one message can ever dovetail with a given message. A thread can only read from

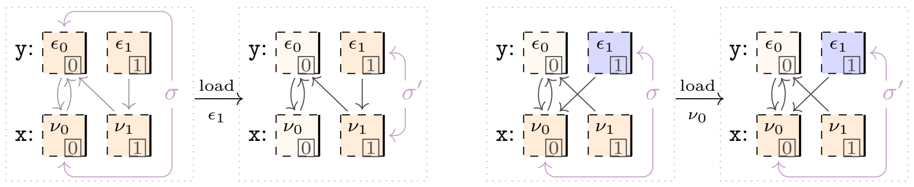


Fig. 2. Depictions of a step during an execution of a litmus test, with the view of the right thread changing from σ to σ'. The value each message carries is in its bottom-right corner. Views are illustrated implicitly in the graph structure that they impose. Obscured messages are faded. **Left:** As the right thread in (MP) loads 1 from y, it inherits the view of ϵ_1, obscuring ν_0. **Right:** The right thread in (SB) loading 0 from x. Storing ϵ_1 did not obscure ν_0.

revealed messages, and when it reads, its view increases as needed to dominate the view of the loaded message. This may obscure messages at other locations.

Revisiting the (MP) litmus test, starting with a memory with a single message holding 0 at each location, and with all views pointing to the timestamps of these message, suppose the right thread loaded 1 from y, as depicted on the left side of Figure 2. Such a message can only be available if the left thread stored it. Before storing 1 to y, the left thread stored 1 to x, obscuring the initial x message. The right thread inherits this limitation through the causal relationship, so it will not be able to load 0 from x. Therefore, RA forbids the outcome $\langle\langle\rangle, \langle 1, 0\rangle\rangle$.

In contrast, consider the litmus test known as *store buffering*:

$$(\mathtt{x} := 1\,;\mathtt{y}?) \parallel (\mathtt{y} := 1\,;\mathtt{x}?) \tag{SB}$$

By considering the possible interleavings, one can check that no execution in SC returns $\langle 0, 0\rangle$. However, in RA some do. Indeed, even if the left thread stores to x before the right thread loads from x, the right thread's view allows it to load 0, as depicted on the right side of Figure 2.

We can recover the SC behavior by interspersing fences between sequenced memory accesses, which we model with FAA$(\mathtt{z}, 0)$ to a fresh location z. Thus, compare (SB) to the *store buffering with fences* litmus test:

$$(\mathtt{x} := 1\,;\mathrm{FAA}\,(\mathtt{z}, 0)\,;\mathtt{y}?) \parallel (\mathtt{y} := 1\,;\mathrm{FAA}\,(\mathtt{z}, 0)\,;\mathtt{x}?) \tag{SB+F}$$

Both of the FAA$(\mathtt{z}, 0)$ instructions store messages that must dovetail with the message that they load from, and in that also inherit its view. They cannot both dovetail with the same message because their segments cannot intersect. Thus, one of them—say, the one on the right—will have to dovetail with the other. In this scenario, the view of the message that the left thread stores at z points to the message it previously stored at x. When the right thread loads the message from z it inherits this view, obscuring the initial message to x. Therefore, when it later loads from x, it must load what the left thread stored. Thus, like in SC, no execution in RA returns $\langle 0, 0\rangle$.

3 Denotational Semantics for Release/Acquire

We start this section by explaining how we support first-class concurrent composition ($\|$) in the operational semantics of Release/Acquire (§3.1). In the rest of the section we present the core of our denotational semantics. First, we present our notion of a trace, adapted to RA, along with four basic rewrite rules that our denotations are closed under (§3.2). Next, we define the denotations of the key program constructs (§3.3). We then present further aspects of the denotational semantics that make it more abstract: restrictions that traces in denotations must uphold (§3.4), and three more rewrite rules under which denotations are closed (§3.5). For completeness, we show how to give denotations to the whole language standardly, using Moggi's approach (§3.6).

3.1 First-class Concurrent Composition

Kang et al. presentation assumes top-level parallelism, a common practice in studies of weak-memory models. This comes at the cost of the uniformity and compositionality. In particular, the denotation $[\![M \parallel N]\!]$ cannot be defined. We resolve this by extending Kang et al.'s operational semantics to support first-class parallelism by organizing thread views in an evolving *view-tree*, a binary tree with view-labelled leaves, rather than in a fixed flat mapping. Thus, *states* that accompany executing terms consist of a memory and a view-tree. In discourse, we do not distinguish between a view-leaf and its label.

An *initial state* consists of a memory with a single message at each location, and a view which points to these messages' timestamps. The example below shows how threads inherit their parent's view upon activation and combine their views as they synchronize:

Example. In the following, $\rightsquigarrow$ is the execution step relation, $\rightsquigarrow^*$ is its reflexive-transitive closure, μ_0 is an initial memory, $\dot{\kappa}$ is the κ-labelled view-leaf, $T \frown R$ is the view-tree that consists of a node connected to the view-trees T and R, and ω is the least view that dominates both ω_1 and ω_2:

$$\langle \mu_0, \dot{\alpha} \rangle, M \mathbin{;} (N_1 \parallel N_2) \rightsquigarrow^* \langle \mu_1, \dot{\alpha}' \rangle, N_1 \parallel N_2 \rightsquigarrow \left\langle \mu_1, \dot{\alpha}' \frown \dot{\alpha}' \right\rangle, N_1 \parallel N_2$$
$$\rightsquigarrow^* \left\langle \rho, \dot{\omega}_1 \frown \dot{\omega}_2 \right\rangle, V_1 \parallel V_2 \rightsquigarrow \langle \rho, \dot{\omega} \rangle, \langle V_1, V_2 \rangle$$

First, M runs until it returns a value, which is discarded by the sequencing construct. Next, the parallel composition $N_1 \parallel N_2$ activates. The threads then interleave executions, each with its associated side of the view-tree. Finally, once both threads return a value, they synchronize.

Handling parallel composition as a first-class construct allows us to decompose Write-Read Reordering (WR-Reord) $(\mathtt{x} := v) \mathbin{;} \mathtt{y?} \twoheadrightarrow \mathbf{fst}\,\langle \mathtt{y?}, (\mathtt{x} := v) \rangle$, a crucial reordering of memory accesses valid under RA but not under SC, into a combination of Write-Read Deorder (WR-Deord) $\langle (\mathtt{x} := v), \mathtt{y?} \rangle \twoheadrightarrow (\mathtt{x} := v) \parallel \mathtt{y?}$ together with structural transformations and laws of parallel programming:

$$
\begin{aligned}
&(\mathtt{x} := v)\ ;\mathtt{y?} \overset{\downarrow\text{Structural}}{\twoheadrightarrow} \mathbf{snd}\ \langle (\mathtt{x} := v), \mathtt{y?} \rangle \overset{\downarrow\text{(WR-Deord)}}{\twoheadrightarrow} \mathbf{snd}\ ((\mathtt{x} := v) \parallel \mathtt{y?}) \\
&\overset{\downarrow\text{Par. Prog. Law: Symmetry}}{\twoheadrightarrow} \mathbf{snd}\ (\mathbf{swap}\ (\mathtt{y?} \parallel (\mathtt{x} := v))) \overset{\downarrow\text{Structural}}{\twoheadrightarrow} \mathbf{fst}\ (\mathtt{y?} \parallel (\mathtt{x} := v)) \overset{\downarrow\text{Par. Prog. Law: Sequencing}}{\twoheadrightarrow} \mathbf{fst}\ \langle \mathtt{y?}, (\mathtt{x} := v) \rangle
\end{aligned}
$$

This provides a separation of concerns: the components of this decomposition are supported by our semantics using independent arguments. It also sheds a light on the interesting part, as they are all valid under SC except for (WR-Deord).

3.2 Traces for Release/Acquire

Adapting Brookes's SC-traces, our RA-traces also include a sequence of transitions ξ, each transition a pair of RA memories; and a return value r. Intuitively, these play a similar role here, formally grounded in analogs to the stutter and mumble rewrite rules. Seeing that the operational semantics only adds messages and never modifies them, we require that every memory snapshot in the sequence ξ be contained in the subsequent one, whether it be within or across transitions. A message added within a transition is a *local message*; otherwise it is an *environment message.* We call the first memory in ξ's first transition its *opening memory*, and the second memory in ξ's last transition its *closing memory.*

In addition, RA-traces include an initial view α, declaring which messages are relied upon to be revealed in ξ's opening memory; and a final view ω, declaring which messages are guaranteed to be revealed in ξ's closing memory. We ground these intuition formally in the rewind and forward rewrite rules below.

We write the trace as $\alpha\boxed{\xi}\omega \mathrel{.\dot{}.} r$. See an illustration on the bottom of Figure 1.

Stutter & Mumble. We define the stutter (St) and mumble (Mu) rewrite rules:

$$
\alpha\boxed{\xi\eta}\omega \therefore r \xrightarrow{\mathsf{St}} \alpha\boxed{\xi\langle\mu,\mu\rangle\eta}\omega \therefore r \qquad \alpha\boxed{\xi\langle\mu,\rho\rangle\langle\rho,\theta\rangle\eta}\omega \therefore r \xrightarrow{\mathsf{Mu}} \alpha\boxed{\xi\langle\mu,\theta\rangle\eta}\omega \therefore r
$$

As in Brookes's semantics, their role is to make the semantics more abstract by divorcing the length of the sequence from the individual steps taken in the operational semantics, while maintaining the transitions' Rely/Guarantee character.

Rewind & Forward. The rewind (Rw) rewrite rules establish the fact that the term only relies on certain messages being *revealed*, not on messages being obscured. The rewind rule modifies the initial view, making it point to earlier messages on the timelines. Thus, relied upon messages will remain available after the rewrite. Similarly, the forward (Fw) rewrite rule establish the fact that the term only guarantees that certain messages are revealed. The forward rule modifies the final view, making it point to later messages on the timelines. Thus, any message guaranteed to be available was already guaranteed beforehand. The rules are schematically depicted in Figure 3.

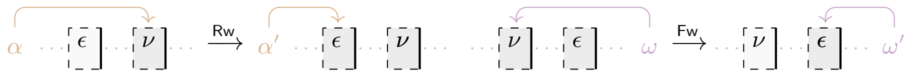


Fig. 3. Schematic depictions of the rewind and forward rewrite rule, focusing on a single location, where the initial/final view points to ν before and points to ϵ after. The messages ν and ϵ may coincide, dovetail, or be separated. **Left:** The initial view α is "rewound" to α'. **Right:** The final view ω is "forwarded" to ω'.

3.3 Introducing Denotations for RA

We present denotations of key constructs of the programming language. By referring to the notion of a *closed set* below, we mean a set that is closed under certain rewrite rules, such as stutter, mumble, rewind, and forward from §3.2.

Pure. A pure (i.e. effect-free) computation guarantees a returned value, and otherwise can only guarantee what it relies on. For example, define $[\![2 + 3]\!]$ as least closed set with all traces of the form $\kappa\boxed{\langle\mu,\mu\rangle}\kappa \mathrel{.\,\because} 5$.

Sequence. In denoting sequential composition we must make sure that the first component does not obscure any message that the second component relies on. Thus, define $[\![\langle M, N\rangle]\!]$ as least closed set with all traces of the form $\alpha\boxed{\xi\eta}\omega \mathrel{.\,\because} \langle r, s\rangle$, where there exists a view κ such that $\alpha\boxed{\xi}\kappa \mathrel{.\,\because} r \in [\![M]\!]$ and $\kappa\boxed{\eta}\omega \mathrel{.\,\because} s \in [\![N]\!]$. The *existence* of the revealed messages is implicit: ξ's closing memory must be contained in the memory that follows it, which is η's opening memory. The definition of $[\![M \mathbin{;} N]\!]$ is the same, except that the first component of the returned pair is discarded. That is, with traces of the form $\alpha\boxed{\xi\eta}\omega \mathrel{.\,\because} s$.

Parallel. Threads composed in parallel rely on the same preceding sequential environment and guarantee to the same succeeding sequential environment. Thus, define $[\![M_1 \parallel M_2]\!]$ as the least closed set with all traces of the form $\alpha\boxed{\xi}\omega \mathrel{.\,\because} \langle r_1, r_2\rangle$, where there exist sequences ξ_1 and ξ_2 such that and ξ is obtained by interleaving their transitions, and $\alpha\boxed{\xi_i}\omega \mathrel{.\,\because} r_i \in [\![M_i]\!]$ (for $i \in \{1, 2\}$).

Dereference. We define $[\![\ell?]\!]$ to be the least closed set with all traces of the form $\alpha\boxed{\langle\mu,\mu\rangle}\omega \mathrel{.\,\because} v$, where $\ell{:}v@(q, \alpha(\ell)]\langle\!\langle\kappa\rangle\!\rangle \in \mu$ for some timestamp q and view κ, and both $\alpha \le \omega$ and $\kappa \le \omega$.

Assignment. Define $[\![\ell := v]\!]$ as the least closed set with all traces of the form $\alpha\boxed{\langle\mu,\rho\rangle}\omega \mathrel{.\,\because} \langle\rangle$ where ρ is obtained by adding the message $\ell{:}v@(q, \omega(\ell)]\langle\!\langle\omega\rangle\!\rangle$ to μ for some timestamp q, and $\alpha \le \omega$.

Read-modify-write. The definition of $[\![\mathrm{FAA}\,(\ell, w)]\!]$ combines the two above, along with a dovetailing requirement. Specifically, it is the least closed set with all traces of the form $\alpha\boxed{\langle\mu,\rho\rangle}\omega \mathrel{.\,\because} v$, where $\ell{:}v@(q, \alpha(\ell)]\langle\!\langle\kappa\rangle\!\rangle \in \mu$ for some timestamp q and view κ, both $\alpha \le \omega$ and $\kappa \le \omega$, and ρ is obtained by adding the message $\ell{:}\,(v{+}w)\,@(\alpha(\ell), \omega(\ell)]\langle\!\langle\omega\rangle\!\rangle$ to μ. The semantics of other RMWs is defined similarly.

Example. We show that $[\![\ell := v \,;\, v]\!] \subseteq [\![\ell := v \,;\, \ell?]\!]$. When sequencing two traces, the final view of the first must match the initial view of the second, so traces in $[\![\ell := v \,;\, v]\!]$ have the form $\alpha\boxed{\langle\mu,\rho\rangle\,\langle\theta,\theta\rangle}\omega \mathrel{.\!\cdot.} v$, where ρ is obtained by adding the message $\ell{:}v@(q, \omega(\ell)]\langle\!\langle\omega\rangle\!\rangle$ to μ for some timestamp q, and $\alpha \leq \omega$. Since ω points to this added message, and since $\rho \subseteq \theta$ as memories along a trace's sequence, $\omega\boxed{\langle\theta,\theta\rangle}\omega \mathrel{.\!\cdot.} v \in [\![\ell?]\!]$. By sequencing, $\alpha\boxed{\langle\mu,\rho\rangle\,\langle\theta,\theta\rangle}\omega \mathrel{.\!\cdot.} v \in [\![\ell := v \,;\, \ell?]\!]$.

3.4 Correspondence to the Operational Semantics

Traces in denotations, if unconstrained, may represent behaviors that include operationally unreachable states. Forbidding such redundant traces eliminates a source of differentiation between denotations, thus increasing their abstraction.

Reachable states. Consider the transformation `x?` ; `y?` $\twoheadrightarrow$ `y?`, a consequence of the RA-valid Irrelevant Read Elimination (R-Elim) `x?` ; $\langle\rangle \twoheadrightarrow \langle\rangle$ and structural equivalences. Consider the state S that consists of the memory at the top of Figure 1 and the view that points to ν_3 and ϵ_2. The only step `x?` ; `y?` can take from the state S is to load ν_3, inheriting the view that ν_3 carries, which changes the thread's view to point to ϵ_3. Only ϵ_3 is available in the following step, which means the term returns 3. In contrast, starting from S, the term `y?` can load from ϵ_2 to return 7. This analysis does not invalidate the transformation because the state S is unreachable by an execution starting from an initial state, and should therefore be ignored when determining observable behaviors.

Internalizing invariants. Just as we ignore unreachable states in the operational semantics, we discard "unreachable" traces to refine our denotational semantics. We consider a state to be valid if it adheres to the following invariants.

Scattering: segments in memory never overlap.
Pointing: views always point to messages.
Dominating: views always dominate the views of the messages to which they point. This invalidates the state S above, because the view of the thread does not dominate the view of ν_3 even though it points to it.
Descending: a path from a message along the view-induced graph structure cannot end in another message with a greater timestamp at the same location. Demonstrated both positively and negatively in Figure 4.
Acyclicity: a cycle along the view-induced graph structure consists solely of messages which have the smallest timestamp on their timeline.

Memory snapshots in traces are required to obey each of the invariants above. The initial and final view must point to and dominate the opening and closing memory respectively. This means that there must be a message to load that allows the initial and final view to be equal, and we obtain $[\![$`x?` ; $\langle\rangle]\!] \supseteq [\![\langle\rangle]\!]$.

We also uphold requirements that correspond to the relation between the states across a possibly-interrupted series of steps in the operational semantics:

Accumulating: the memory after contains the memory before. We require that every memory snapshot contains the one before it.

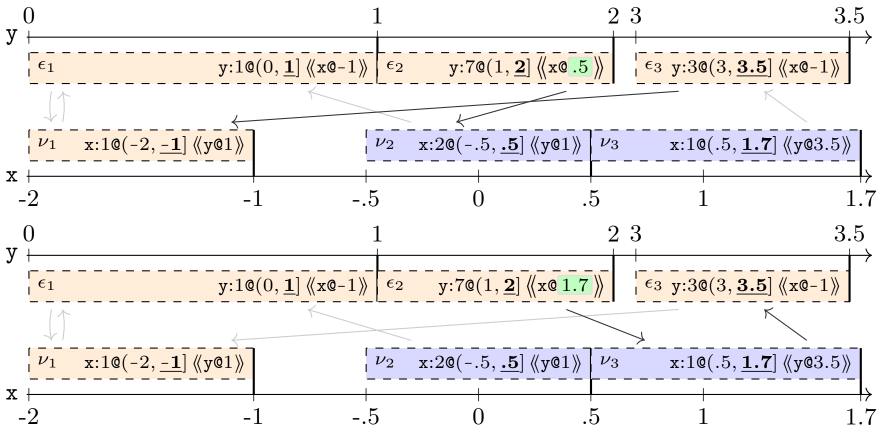


Fig. 4. Two variations on the memory illustrated in Figure 1. **Top:** This can function as a memory snapshot in a trace. It demonstrates that the views of messages along a timeline do not have to be ordered: ϵ_2 appears earlier than ϵ_3 on y's timeline but points to a later message on x's timeline. **Bottom:** This cannot function as a memory snapshot in a trace, because it contains an ascending path. Intuitively, no thread could have written ϵ_2 because the view that ϵ_2 carries indicates that the thread would have already "known" about ν_3 and therefore, following the causality chain, about ϵ_3 as well. Thus, the thread would have been forbidden from picking ϵ_2's timestamp.

Delimiting: if the view-trees before and after are leaves, then the view after dominates the view before, and the view of any written message dominates the view before and is dominated by the view after. We impose the analogous requirement on the initial and final views, and on the local messages.

The trace in Figure 1 adheres to the invariants and relationships we have listed.

Concrete operational correspondence. We call the rewrite rules that were defined in §3.2 *concrete* because they maintain a certain concrete interpretation of traces. To see this, consider the operational semantics for RA augmented with an additional kind of step, which any term can take. The only change along this step is that a view in the view-tree inherits the view from a message that is available to it. This addition does not change the observable behaviors of whole programs, and maintains the above invariants.

Each trace in the denotations of §3.3, if closed only under the concrete rewrite rules, corresponds to an interrupted execution in the augmented operational semantics. The correspondence is similar to that from Brookes's semantics in terms of the sequence of transitions and return value. The initial and final views determine the views at the beginning and the end of the interrupted execution.

The introduction of the rewrite rules in §3.5 will mean that traces do not have such a clear operational interpretation. The key to our proof of adequacy

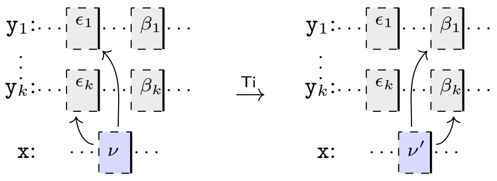


Fig. 5. Schematic depiction of the tighten rewrite rule, that focuses on a particular memory snapshot within the trace, in the setting of $k+1$ locations. The message ν is "tightened" to ν', such that for each i it points to β_i instead of ϵ_i. This includes the case that β_i and ϵ_i are the same message in some locations.

is to partially recover this operational correspondence in terms of the overall observable behaviors (§4).

3.5 Abstract Rewrite Rules

Transitions in RA traces consist of sets of messages, which record much more information about the operational execution than the mappings from locations to values we had in SC. This makes the trace-based semantics too concrete. We resolve the memory-concreteness issue by introducing three *abstract* rewrite rules that obfuscate information about local messages. This makes the denotations more abstract by blurring the distinctions that denotations can make.

Tighten. Recall the transformation (WR-Deord) that we wish to support. Let $\tau_1 \in [\![\mathtt{x} := v]\!]$ and $\tau_2 \in [\![\mathtt{y?}]\!]$, such that they compose sequentially to form a trace from $[\![\langle(\mathtt{x} := v), \mathtt{y?}\rangle]\!]$. Then τ_1's final view κ must equal τ_2's initial view. The view κ dominates the view σ of the local message ν_1 stored by τ_1, and κ cannot obscure the message ν_2 from which τ_2 loaded its value. Thus, σ cannot obscure ν_2. In contrast, consider τ_1 and τ_2 that compose in parallel to form a trace from $[\![(\mathtt{x} := v) \parallel \mathtt{y?}]\!]$. Here, the view of the local message may very well obscure the loaded message. Indeed, the final view of τ_1 may dominate the initial view of τ_2.

To resolve this, observe that the purpose of recording views in messages is to encumber its loaders. Under this perspective, the view of a local message guarantees to the environment that loading the local message will keep certain messages revealed. Therefore, making the view larger only weakens the guarantee. Thus, we introduce the tighten (Ti) rewrite rule that makes the view of a local message larger. The rule is depicted in Figure 5, and Figure 6 provides a concrete example. Using tighten, we can show that $[\![\langle(\mathtt{x} := v), \mathtt{y?}\rangle]\!] \supseteq [\![(\mathtt{x} := v) \parallel \mathtt{y?}]\!]$.

Absorb. Recall the transformation (WW-Elim) that we wish to support. To show this we aim to replicate, as far as we can, the reasoning we have used to show $[\![\mathtt{x} := v\,; \mathtt{x} := w]\!] \supseteq [\![\mathtt{x} := w]\!]$ in Brookes's semantics. Recall that, to use mumble, we made the memories match across the two transitions of $[\![\mathtt{x} := v\,; \mathtt{x} := w]\!]$. Doing so here, we end up with two local messages, whereas traces from $[\![\mathtt{x} := w]\!]$ only have a single local message. Roughly speaking, the equality concerning SC

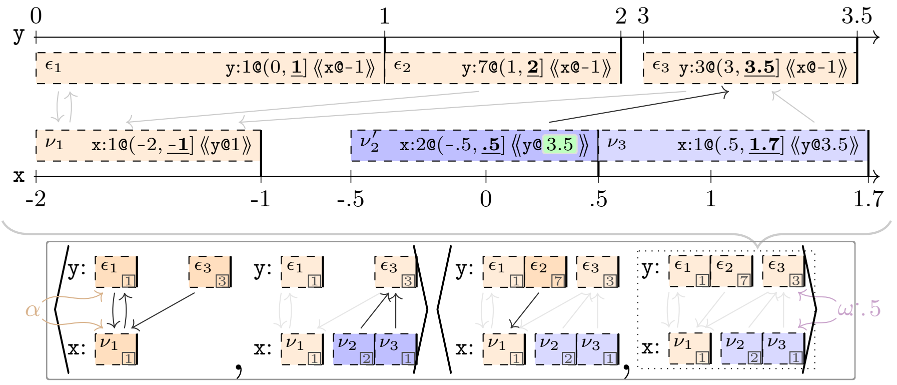


Fig. 6. A possible result from rewriting the trace from Figure 1 using tighten. Since ν_2 is local in the trace from Figure 1, tighten can advance its view to point to ϵ_3 instead of ϵ_1. The same replacement is applied throughout the trace's sequence, not just the closing memory.

memories $\mu\,[\mathtt{x} := v]\,[\mathtt{x} := w] = \mu\,[\mathtt{x} := w]$ does not transfer to RA where memory, by accumulating messages, is more concrete. We resolve this by adding the absorb (Ab) rewrite rule, which replaces two dovetailed local messages with one that carries the second message's value. The rule is depicted in Figure 7, and Figure 8 provides a specific example.

Dilute. There is another known family of transformations that are valid under RA memory, yet we cannot justify with the rules we presented. These introduce non-modifying atomic updates, such as Read to FAA (R-FAA) $\ell\mathbf{?} \twoheadrightarrow \mathrm{FAA}\,(\ell, 0)$.

Running within some context, $\mathrm{FAA}\,(\ell, 0)$ reads a message ν, to which it dovetails another message ϵ with the same value. It's possible that some β dovetails with ϵ later in the execution. In the same context, we can simulate this behavior with $\ell\mathbf{?}$ instead, by having the context provide ν' instead of ν, with the difference that it takes up the same segment that ν and ϵ have taken up combined. If there is a β as mentioned, it can now dovetail with ν' to the same effect. In this scenario, ν is an environment message, but we must also account for the case that it is local to allow for composition, such as in $\ell\mathbf{:=}v\,\mathbf{;}\,\ell\mathbf{?} \twoheadrightarrow \ell\mathbf{:=}v\,\mathbf{;}\,\mathrm{FAA}\,(\ell, 0)$.

We internalize the idea behind this argument as the dilute (Di) rewrite rule, in which a message is replaced by two message that together occupy the same segment, the second being a local message that cannot appear before the first in the trace and must carry the same value. With dilute, $[\![\ell\mathbf{?}]\!] \supseteq [\![\mathrm{FAA}\,(\ell, 0)]\!]$. The rule is depicted in Figure 7, and Figure 9 provides a specific example.

3.6 Monadic Presentation

One of the contributions of this work is to bridge research of weak-memory models with Moggi's monad-based approach [38] to denotational semantics. In this approach, one start by defining a monad, which has three components. The

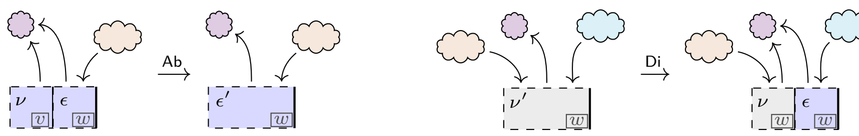


Fig. 7. Schematic depictions of the absorb (left) and dilute (right) rewrite rules, that focus on the segment of the dovetailed messages together with all pointers into and out of them, within a particular memory snapshot. The *circular* cloud represents the subset of the memory that the messages in focus are pointing to, showing that they all have the same view. The *elliptical* clouds represent views—including the initial and final view, as well as other messages—that point to each of the dovetailing messages. **Left:** The message ν is "absorbed" into the message ϵ to become ϵ'. No view may point to ν. **Right:** The message ν' "dilutes" into ν and ϵ. While ϵ must be a local message, ν and ν' can appear anywhere the trace's sequence, as long as they appear in the same places in the sequence, and that ϵ does not appear before. The views that point to ν' before diluting can point either to ν or to ϵ after diluting.

first associates for every set X, which we think of as representing returned values, to a set $\underline{\mathcal{T}}X$ representing computations that return values from X. In our case, $\underline{\mathcal{T}}X$ consists of countable sets of traces closed under rewrite rules.

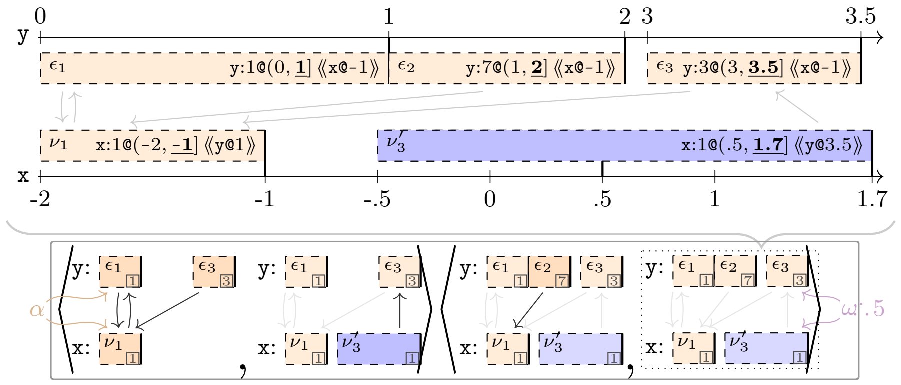
Denotations are then defined according to their *typing judgments*. For example, $a, b : \mathsf{Loc} \vdash \langle a, b\mathbf{?}\rangle : (\mathsf{Loc} \times \mathsf{Val})$ means that in the context that the free variables a and b are locations, the term $\langle a, b\mathbf{?}\rangle$ is a location-value pair. Given a function γ that maps a and b to locations, $[\![\langle a, b\mathbf{?}\rangle]\!]\,\gamma \in \underline{\mathcal{T}}(\mathsf{Loc} \times \mathsf{Val})$. For $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, we generalize containment $[\![N]\!] \supseteq [\![M]\!]$ pointwise: if γ maps variables in Γ appropriately by their type, then $[\![N]\!]\,\gamma \supseteq [\![M]\!]\,\gamma$. This degenerates when Γ is empty, i.e. when M and N are *closed terms*.

The second monad component is a function $\mathrm{return}^{\mathcal{T}}_{X} : X \to \underline{\mathcal{T}}X$ maps values to pure computations that return that value. The third component sequences computations, such that the latter depends on the value returned by the former: $(\rangle\!\rangle\!=^{\mathcal{T}}_{X,Y}) : (\underline{\mathcal{T}}X) \times (X \to \underline{\mathcal{T}}Y) \to \underline{\mathcal{T}}Y$. Omitting the indices, the monad components must satisfy certain axioms that formalize the stated intuition: $\mathrm{return}\,r \rangle\!\rangle\!= f = f(r)$, $P \rangle\!\rangle\!= \mathrm{return} = P$ and $(P \rangle\!\rangle\!= f) \rangle\!\rangle\!= g = P \rangle\!\rangle\!= \lambda r.\,(f(r) \rangle\!\rangle\!= g)$.

In our case, we define $\mathrm{return}\,r$ as the least closed set with all traces of the form $\kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore r$; and $P \rangle\!\rangle\!= f$ as the least closed set with all traces of the form $\alpha\boxed{\xi\eta}\omega \therefore s$, where $\alpha\boxed{\xi}\kappa \therefore r \in P$ and $\kappa\boxed{\eta}\omega \therefore s \in f(r)$ for some κ.

Denotations. This approach comes read-made with denotations for standard language constructs. For example, $[\![\langle M, N\rangle]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \rangle\!\rangle\!= \lambda r.\,([\![N]\!]\,\gamma \rangle\!\rangle\!= \lambda s.\,\langle r, s\rangle)$. Similarly, $[\![\mathbf{match}\,M\,\mathbf{with}\,\langle a, b\rangle.\,N]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \rangle\!\rangle\!= \lambda\langle r, s\rangle.\,[\![N]\!]\,\gamma\,[a \mapsto r]\,[b \mapsto s]$, where $\gamma\,[a \mapsto r]$ is obtained from γ by mapping a to r. Pure computations use the return function, e.g. $[\![v]\!] = \mathrm{return}\,v$.

Program effects can be modularly introduced in this approach, such as memory access, where $[\![\ell \coloneqq v]\!] \in \underline{\mathcal{T}}\,\{\langle\rangle\}$ and $[\![\ell\mathbf{?}]\!], [\![\mathrm{FAA}\,(\ell, v)]\!] \in \underline{\mathcal{T}}\mathsf{Val}$; and par-



0
1
2
3
3.5
y
ϵ_1 y:1@(0, 1] ⟨⟨x@-1⟩⟩
ϵ_2 y:7@(1, 2] ⟨⟨x@-1⟩⟩
ϵ_3 y:3@(3, 3.5] ⟨⟨x@-1⟩⟩
ν_1 x:1@(-2, -1] ⟨⟨y@1⟩⟩
ν_3' x:1@(.5, 1.7] ⟨⟨y@3.5⟩⟩
x
-2
-1
-.5
0
.5
1
1.7
α
ω:.5

Fig. 8. A possible result from rewriting of the trace from Figure 6 using absorb. The dovetailed messages ν_2 and ν_3 are local in the trace from Figure 1, added within the same transition, so by rewriting by absorb they can be replaced by ν_3' obtained by stretching ν_3's segment to cover ν_2's segment.

allel composition, a function $(|||^{\mathcal{T}}_{X,Y}) : \underline{\mathcal{T}}X \times \underline{\mathcal{T}}Y \to \underline{\mathcal{T}}(X \times Y)$ with which $[\![M \parallel N]\!]\,\gamma \coloneqq [\![M]\!]\,\gamma \mathbin{|||} [\![N]\!]\,\gamma$. The definition remains the same: we obtain traces in $P \mathbin{|||} Q$ by interleaving transitions and pairing returned values of traces with matching views, one from P and one from Q.

Adhering to left-to-right evaluation both operationally and denotationally, $M \mathbin{\boldsymbol{:=}} N$ is equivalent to $\mathbf{match}\,\langle M, N\rangle\,\mathbf{with}\,\langle a, b\rangle.\, a \mathbin{\boldsymbol{:=}} b$. In traces of assignment, the added local message is free to dovetail with a previous message, unlike in RMW traces where it must. Therefore, we have $[\![\ell \mathbin{\boldsymbol{:=}} (\ell\boldsymbol{?} + v)]\!] \supseteq [\![\mathrm{FAA}\,(\ell, v)]\!]$.

Structural reasoning. Among the general results and proof techniques this approach supplies are *structural equivalences.* These are denotational equations that hold due to the properties of the core calculus, and are preserved by modular expansions with program effects. For instance, if K is effect-free, then $[\![\mathbf{if}\,K\,\mathbf{then}\,M\,\boldsymbol{;}\,N\,\mathbf{else}\,M\,\boldsymbol{;}\,N'\,]\!] = [\![M\,\boldsymbol{;}\,\mathbf{if}\,K\,\mathbf{then}\,N\,\mathbf{else}\,N'\,]\!]$. Equivalences such as this one may otherwise require challenging ad-hoc proofs [e.g. 24, 26].

More generally, structural reasoning composes to derive further equivalences. For example, from $[\![\langle\rangle]\!] = [\![\ell\boldsymbol{?}\,\boldsymbol{;}\,\langle\rangle]\!]$ and structural equivalences, namely "left neutrality" $[\![K]\!] = [\![\langle\rangle\,\boldsymbol{;}\,K]\!]$ and "associativity" $[\![(M\,\boldsymbol{;}\,N)\,\boldsymbol{;}\,K]\!] = [\![M\,\boldsymbol{;}\,(N\,\boldsymbol{;}\,K)]\!]$:

$$[\![K]\!] = [\![\langle\rangle\,\boldsymbol{;}\,K]\!] = [\![(\ell\boldsymbol{?}\,\boldsymbol{;}\,\langle\rangle)\,\boldsymbol{;}\,K]\!] = [\![\ell\boldsymbol{?}\,\boldsymbol{;}\,(\langle\rangle\,\boldsymbol{;}\,K)]\!] = [\![\ell\boldsymbol{?}\,\boldsymbol{;}\,K]\!] \qquad (\star)$$

Structural reasoning generalizes to program transformations. For example, $(\rangle\!\rangle\!=)$ is monotonic, so we can also derive:

$$[\![\langle\rangle]\!] = [\![\ell\boldsymbol{?}\,\boldsymbol{;}\,\langle\rangle]\!] = [\![\ell\boldsymbol{?}]\!]\rangle\!\rangle\!=\!\lambda v.[\![\langle\rangle]\!] \supseteq [\![\mathrm{FAA}\,(\ell, 0)]\!]\rangle\!\rangle\!=\!\lambda v.[\![\langle\rangle]\!] = [\![\mathrm{FAA}\,(\ell, 0)\,\boldsymbol{;}\,\langle\rangle]\!]$$

Since $(|||)$ is also monotonic, we can use this to show that $[\![(\mathrm{SB})]\!] \supseteq [\![(\mathrm{SB{+}F})]\!]$.

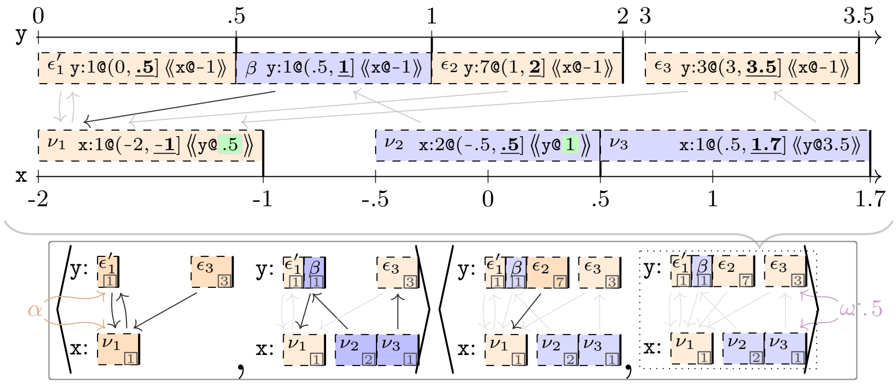


Fig. 9. A possible result from rewriting of the trace from Figure 1 using dilute. The message ϵ_1 from Figure 1 was replaced with ϵ_1', with the same value 1. The local message β—which takes up the rest of the missing space left behind by ϵ_1—always appears with ϵ_1', dovetailing with it and carrying the same value. The message ϵ_2, that used to dovetail with ϵ_1, now dovetails with β.

Higher order. An important aspect of a programming language is its facilitation of abstraction. Higher-order programming is a flexible instance of this, in which programmable functions can take functions as input and return functions as output. Moggi's approach supports this feature out-of-the-box, in such a way that does not complicate the rest of the semantics, as the first-order fragment of the semantics need not change to include it.

Every value returned by an execution has a semantic presentation which we use as the returned value in traces. The semantic and syntactic values are identified in the first-order fragment, but different syntactic functions may have the same semantics, so the identification does not extend to higher-order.

We classify a term as a *program* if it is *closed* (every variable occurrence is bound) and of *ground type* (all functions are applied to arguments). This definition is in line with the expectation that a program should return a concrete result that the end-user can consume. Thus, we only consider observable behaviors of programs. Transformations only need to be valid when applied within programs. Programs degenerate to closed terms in the first-order fragment.

4 Main Results

We present the main results that we have proven about our denotational semantics. Moggi's semantic toolkit features ubiquitously in their proofs.

Compositionality. In its most basic form, this key feature of denotational semantics means that a program term's denotation is defined using *the denotations* of its immediate subterms. We have used this in ($\star$). In our case denotations are sets, where each elements represents a possible behavior of the term, we are interested in establishing a directional generalization of compositionality:

Lemma 1. *If* $[\![M]\!] \subseteq [\![N]\!]$ *then* $[\![\Xi[M]]\!] \subseteq [\![\Xi[N]]\!]$ *for any program context* $\Xi[-]$.

Compositionality is a consequence of its monadic design using monotonic operators, and is not substantially different from previous work [e.g. 20].

Observability correspondence. The abstract rewrite rules break the direct correspondence between traces and interrupted executions. For example, in our analysis of (WW-Elim), by using absorb, we ended up with a trace in which only one message is added even though the program term adds two messages.

Still, some connection must remain to obtain a proof of adequacy. In particular, we would like traces to correspond to observable behavior of programs. In one direction, an even stronger property holds, known as soundness:

Lemma 2. *For every execution of a program* M *in the operational semantics of RA, there exists* $\alpha\boxed{\langle\mu,\rho\rangle}\omega \mathrel{.\cdot.} r \in [\![M]\!]$ *that matches the execution:* $\langle\alpha,\mu\rangle$ *is the initial state,* $\langle\omega,\rho\rangle$ *is the final state, and* r *matches the value returned.*

To prove soundness, we take a trace where transitions correspond to the memory-accessing execution steps, and then use mumble to obtain a single transition.

Ignoring the final state, the correspondence holds in the other direction too:

Lemma 3. *For every program* M *and* $\alpha\boxed{\langle\mu,\rho\rangle}\omega \mathrel{.\cdot.} r \in [\![M]\!]$ *there is an observable behavior of* M *with initial state* $\langle\alpha,\mu\rangle$ *and return value matching* r.

The lack of correspondence with the final state is an artifact of the concreteness-abstraction divergence between the operational and denotational semantics. Due to this divergence, it is significantly more challenging to establish this direction of the correspondence than in previous work.

Overcoming the concreteness-abstraction hurdle. The most technically challenging step in proving Lemma 3 is to prove the application of abstract rewrite rules can be deferred to the end. We define the *basic denotation* of a term M by $\underline{[\![M]\!]}$, which is the denotation were it defined using only the concrete rewrite rules. Denoting its closure under the abstract rewrite rules by $\underline{[\![M]\!]}^{\dagger}$, we claim:

Lemma 4. *If* M *is a program, then* $\underline{[\![M]\!]}^{\dagger} = [\![M]\!]$.

Thus, to obtain all of the traces that result from the regular denotational construction, where all of the rewrite rules are applied throughout the entire denotational construction, it is enough to close only under the concrete rewrite rules as the denotation of a program is built-up from its subterms, applying the abstract rewrite rules only at the top level.

The intuition that guides the inductive proof of Lemma 4 is that the abstract rewrite rules can be percolated out. To get the main idea across while keeping the discussion self-contained, we focus on the $\underline{[\![M_1 \parallel M_2]\!]}^{\dagger} \supseteq [\![M_1 \parallel M_2]\!]$ case.

Let $\pi \in [\![M_1 \parallel M_2]\!]$. By definition, π is obtained by first composing some $\tau_1 \in [\![M_1]\!]$ in parallel with some $\tau_2 \in [\![M_2]\!]$, i.e. interleaving transitions and pairing return values, and then rewriting the resulting trace τ with concrete and abstract rules. By the inductive hypothesis, $\underline{[\![M_i]\!]}^\dagger \supseteq \underline{[\![M_i]\!]}$. So $\tau_i \in \underline{[\![M_i]\!]}^\dagger$, meaning that τ_i is the result of rewriting some $\tau_i' \in \underline{[\![M_i]\!]}$ with abstract rules.

To warm up, we first address the case where $\tau_1' \xrightarrow{\mathsf{Ab}} \tau_1$ and $\tau_2' = \tau_2$. We would hope, naively, that we can compose τ_1' with τ_2' to obtain some $\tau' \in \underline{[\![M_1 \parallel M_2]\!]}$ such that $\tau' \xrightarrow{\mathsf{Ab}} \tau$, and thus τ' rewrites to π. However, they do not compose because τ_1' has two local message, and τ_2' has only the one environment message that matches the result of "absorbing" the two messages. Rather, τ_1' can compose with a trace $\bar{\tau}_2$ which is equal to τ_2' except for having the required two environment messages instead of the combined one.

We formalize this by introducing a dual auxiliary rewrite rule $\bar{\mathsf{x}}$ for each abstract rule x. For example, the dual of absorb is expel, which splits up an environment message dually to how absorb combines local messages. The auxiliary rewrite rules keep us within the basic denotations:

Lemma 5. *If $\tau \in \underline{[\![M]\!]}$ and $\tau \xrightarrow{\mathsf{z}} \pi$ for some auxiliary rule z, then $\pi \in \underline{[\![M]\!]}$.*

Then we apply $\tau_2' \xrightarrow{\bar{\mathsf{x}}} \bar{\tau}_2 \in \underline{[\![M_i]\!]}$, and obtain the required τ' by composing τ_1' in parallel with $\bar{\tau}_2$. This process of applying the dual rewrite in order to percolate an abstract rewrite out holds for sequential composition too. We summarize:

Lemma 6. *If $\pi' \xrightarrow{\mathsf{x}} \pi$ for some abstract x, and π composes in parallel with ϱ to obtain τ, then there exist $\varrho' \xrightarrow{\bar{\mathsf{x}}} \varrho$ and $\tau' \xrightarrow{\mathsf{x}} \tau$, such that π' composes in parallel with ϱ' to obtain τ'. Similarly for sequential composition.*

In the case where there are more abstract rewrite rules needed to obtain τ_1 from τ_1', we can repeat the process. Yet two problems remain.

The first problem is that π is obtained from $\tau' \in \underline{[\![M_1 \parallel M_2]\!]}$ by both concrete and abstract rewrites, starting with the abstract rewrites that we have "peeled off" τ_1. To show that $\pi \in \underline{[\![M_1 \parallel M_2]\!]}^\dagger$, we need the concrete rewrites to come before the abstract rewrites.

The second problem appears once we remove our simplifying assumption that $\tau_2' = \tau_2$. In the general case, we obtain $\bar{\tau}_2$ from τ_2' using abstract rewrites followed by auxiliary rewrites. If we could replace the sequence of rewrites with one in which the abstract rewrites follow the auxiliary rewrites, then τ_2' could be rewritten with auxiliary rules to some $\bar{\tau}_2' \in \underline{[\![M_2]\!]}$ by using Lemma 5, which in turn could be rewritten with abstract rewrites to $\bar{\tau}_2 \in \underline{[\![M_2]\!]}^\dagger$. This would allow the proof to continue by repeating the process to the other side.

Both problems are solved by commuting the abstract rewrites outwards:

Lemma 7. *For any rewrite sequence starting with τ and ending with π, there exists one in which all of the abstract rewrites appear last.*

Thus, we can do as we planned and repeat the process to the other side, "peeling off" the abstract rewrites from $\bar{\tau}_2$ to obtain $\bar{\tau}_2' \in \underline{[\![M_2]\!]}$, rewriting τ_1' with the dual auxiliary rules in lockstep, resulting in some $\bar{\tau}_1' \in \underline{[\![M_1]\!]}$ by Lemma 5. By Lemma 6, these compose in parallel to some $\bar{\tau} \in \underline{[\![M_1 \parallel M_2]\!]}$ that rewrites with concrete and abstract rules to τ, and thus to π. By Lemma 7, we can rewrite $\bar{\tau}$ with concrete rules to some $\bar{\tau}' \in \underline{[\![M_1 \parallel M_2]\!]}$ first, and with abstract rules afterwards, obtaining $\pi \in \underline{[\![M_1 \parallel M_2]\!]}^{\dagger}$.

Having established Lemma 4, the rest is relatively straightforward. First, traces in basic denotations correspond to interrupted executions, and in particular, an analog of Lemma 3 holds for basic denotations:

Lemma 8. *For every program M and $\alpha\boxed{\langle\mu,\rho\rangle}\omega \therefore r \in \underline{[\![M]\!]}$ there is an observable behavior of M with initial state $\langle\alpha,\mu\rangle$ and return value matching r.*

Next, it is clear from their definition that the abstract rules do not change the number of transitions. Thus, thanks to Lemma 4, the single-transition traces in $[\![M]\!]$ are the result of rewriting single-transition traces in $\underline{[\![M]\!]}$ by abstract rules, which correspond to observable behaviors of M by Lemma 8.

Lemma 3 follows from the fact that the abstract rules preserve the correspondence between traces and observable behavior of programs. For example, due to absorb there is a trace which only adds one message in the denotation of a program that adds two messages; yet the initial view, the opening memory, and the returned value are maintained. The $\mathsf{tighten}$ rule similarly preserves these. In both cases, the execution exhibiting the behavior can remain unchanged. The dilute rule may replace an initial message's timestamp with a smaller one, in which case the execution exhibiting the behavior needs to use the new timestamp accordingly, but otherwise remains the same.

Adequacy. The central result is (directional) adequacy, stating that denotational approximation corresponds to refinement of observable behaviors:

Theorem 9. *If $[\![M]\!] \subseteq [\![N]\!]$, then for all program contexts $\Xi[-]$, every observable behavior of $\Xi[M]$ is an observable behavior of $\Xi[N]$.*

In particular, $[\![M]\!] \subseteq [\![N]\!]$ implies that $N \twoheadrightarrow M$ is valid under RA, because the effect of applying it is unobservable.

Adequacy follows immediately from the above results. Indeed, using soundness, an observable behavior of $\Xi[M]$ corresponds to a single-transition $\tau \in [\![\Xi[M]]\!]$; by the assumption and compositionality $\tau \in [\![\Xi[N]]\!]$; and using the other direction, τ corresponds to an observable behavior of $\Xi[N]$.

Higher-order subtleties. When applying the above results in the presence of higher order, one must pay attention to the *program* assumption. Indeed, suppose $[\![M]\!] \supseteq [\![M']\!]$. Compositionality does not entail that $[\![\lambda a.\, M]\!] \supseteq [\![\lambda a.\, M']\!]$. Indeed, a function $\lambda a.\, M$ is a value, i.e. it does not execute, and in particular it does not perform any effects, regardless of M. Accordingly, $[\![\lambda a.\, M]\!]$ consists of closures of traces of the form $\kappa\boxed{\langle\mu,\mu\rangle}\kappa \therefore f$, where f is a function that returns sets of traces obtained from $[\![M]\!]$. The fact that $[\![M]\!] \supseteq [\![M']\!]$ is not helpful, because traces in $[\![\lambda a.\, M']\!]$ have different returned values f' from traces in $[\![\lambda a.\, M]\!]$.

Directional compositionality is still useful in the presence of abstractions. For example, if M is a program that returns a location, then from $[\![a := v\, ;\, a := w]\!] \supseteq [\![a := w]\!]$ it follows that $[\![(\lambda a.\, a := v\, ;\, a := w)\, M]\!] \supseteq [\![(\lambda a.\, a := w)\, M]\!]$.

Laws of Parallel Programming				
Symmetry	$M \parallel N$	$\twoheadrightarrow$	$\mathbf{swap}\ (N \parallel M)$	
Generalized Sequencing				
	$(\mathbf{let}\, a = M_1\, \mathbf{in}\, M_2) \parallel (\mathbf{let}\, b = N_1\, \mathbf{in}\, N_2)$	$\twoheadrightarrow$	$\mathbf{match}\, M_1 \parallel N_1\, \mathbf{with}\, \langle a, b \rangle.\, M_2 \parallel N_2$	
Eliminations				
Irrelevant Read	$\ell?\, ;\, \langle\rangle$	$\twoheadrightarrow$	$\langle\rangle$	
Write-Write	$\ell := v\, ;\, \ell := w$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\ell := w$	
Write-Read	$\ell := v\, ;\, \ell?$	$\twoheadrightarrow$	$\ell := v\, ;\, v$	
Write-FAA	$\ell := v\, ;\, \mathrm{FAA}\,(\ell, w)$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\ell := (v + w)\, ;\, v$	
Read-Write	$\mathbf{let}\, a = \ell?\, \mathbf{in}\, \ell := (a + v)\, ;\, a$	$\twoheadrightarrow$	$\mathrm{FAA}\,(\ell, v)$	
Read-Read	$\langle \ell?, \ell? \rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \ell?\, \mathbf{in}\, \langle a, a \rangle$	
Read-FAA	$\langle \ell?, \mathrm{FAA}\,(\ell, v) \rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v)\, \mathbf{in}\, \langle a, a \rangle$	
FAA-Read	$\langle \mathrm{FAA}\,(\ell, v)\, , \ell? \rangle$	$\twoheadrightarrow$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v)\, \mathbf{in}\, \langle a, a + v \rangle$	
FAA-FAA	$\langle \mathrm{FAA}\,(\ell, v)\, , \mathrm{FAA}\,(\ell, w) \rangle$	$\overset{\mathsf{Ab}}{\twoheadrightarrow}$	$\mathbf{let}\, a = \mathrm{FAA}\,(\ell, v + w)\, \mathbf{in}\, \langle a, a + v \rangle$	
Others				
Irrelevant Read Introduction	$\langle\rangle$	$\twoheadrightarrow$	$\ell?\, ;\, \langle\rangle$	
Read to FAA	$\ell?$	$\overset{\mathsf{Di}}{\twoheadrightarrow}$	$\mathrm{FAA}\,(\ell, 0)$	
Write-Read Deorder	$\langle (\ell := v)\, , \ell'? \rangle$	$\overset{\mathsf{Ti}}{\twoheadrightarrow}$	$(\ell := v) \parallel \ell'?$	$(\ell \neq \ell')$
Write-Read Reorder	$(\ell := v)\, ;\, \ell'?$	$\overset{\mathsf{Ti}}{\twoheadrightarrow}$	$\mathbf{fst}\ \langle \ell'?, (\ell := v) \rangle$	$(\ell \neq \ell')$

Fig. 10. A selective list of supported non-structural transformations. Along with Symmetry, the denotational semantics supports all symmetric-monoidal laws with the binary operator $(\parallel)$ and the unit $\langle\rangle$. Similar transformations, replacing FAA with other RMWs, are supported too. The abstract rewrite rules used to validate a transformation is mentioned, if there is one.

To deal with the need to prove properties "pointwise" that abstractions bring about, such as containment of denotations in the proof of directional compositionality, we use logical relations. Moggi's toolkit provides a standard way to define these, thereby lifting properties to their higher-order counterparts.

Transformations exhibiting abstraction. To the best of our knowledge, all transformations $N \twoheadrightarrow M$ proven to be valid under RA in the existing literature are supported by our denotational semantics, i.e. $[\![N]\!] \supseteq [\![M]\!]$. Structural transformations are supported by virtue of using Moggi's standard semantics. Our semantics also validates "algebraic laws of parallel programming", such as sequencing $M \parallel N \twoheadrightarrow \langle M, N \rangle$ and its generalization that Hoare and van Staden [22] recognized, $(M_1\, ;\, M_2) \parallel (N_1\, ;\, N_2) \twoheadrightarrow (M_1 \parallel N_1)\, ;\, (M_2 \parallel N_2)$, which in the functional setting can take the more expressive form in which the values returned are passed on to the following computation. See Figure 10 for a partial list.

Hence we claim that our adequate denotational semantics is sufficiently abstract. This supports the case that Moggi's semantic toolkit can successfully scale to handle the intricacies of RA concurrency by adapting Brookes's traces.

5 Related Work and Concluding Remarks

Our work follows the approach of Brookes [13] and its extension to higher-order functions using monads by Benton et al. [6]. Brookes developed a denotational semantics for shared memory concurrency under standard sequentially consistency [33], and established full abstraction w.r.t. a language that has a global atomic **await** instruction that locks the entire memory. The concepts behind this approach had been used in multiple related developments, e.g. [12, 34, 35, 46]. We hope that our work that targets RA will pave the way for similar continuations.

Jagadeesan et al. [25] adapted Brookes's semantics to the x86-TSO memory model [40]. They showed that for x86-TSO it suffices to include the final store buffer at the end of the trace and add two additional simple closure rules that emulate non-deterministic propagation of writes from store buffers to memory, and identify observably equivalent store buffers. The x86-TSO model, however, is much closer to sequential consistency than RA, which we study in this paper. In particular, unlike RA, x86-TSO is "multi-copy-atomic" (writes by one thread are made globally visible to *all* other threads at the same time) and successful RMW operations are immediately globally visible. Additionally, the parallel composition construct in Jagadeesan et al. [25] is rather strong: threads are forked and joined only when the store buffers are empty. Being non-multi-copy-atomic, RA requires a more delicate notion of traces and closure rules, but it has more natural meta-theoretic properties, which one would expect from a programming language concurrency model: sequencing, a.k.a. thread-inlining, is unsound under x86-TSO [see 25, 31] but sound under RA (see Figure 10).

Burckhardt et al. [14] developed a denotational semantics for hardware weak memory models (including x86-TSO) following an alternative approach. They represent sequential code blocks by sequences of operations that the code performs, and close them under certain rewrite rules (reorderings and eliminations) that characterize the memory model. This approach does not validates important optimizations, such as Read-Read Elimination. Moreover, unlike x86-TSO, RA cannot be characterized by rewrite operations on SC traces [31].

Dodds et al. [19] developed a fully abstract denotational semantics for RA, extended with fences and non-atomic accesses. Their semantics is based on RA's *declarative* (a.k.a. axiomatic) formulation as acyclicity criteria on execution graphs. Roughly speaking, their denotation of code blocks (that they assume to be sequential) quantifies over all possible context execution graphs and calculates for each context the "happens-before" relation between context actions that is induced by the block. They further use a finite approximation of these histories to atomically validate refinement in a model checker. While we target RA as well, there are two crucial differences between our work and Dodds et al. [19]. First, we employ Brookes-style totally ordered traces and use interleaving-based operational presentation of RA. Second, and more importantly, we strive for a compositional semantics where denotations of compound programs are defined as functions of denotations of their constituents, which is not the case for Dodds et al. [19]. Their model can nonetheless validate transformations by checking them locally without access to the full program.

Others present non-compositional techniques and tools to check refinement under weak memory models between whole-thread sequential programs that apply for any concurrent context. Poetzl and Kroening [43] considered the SC-for-DRF model, using locks to avoid races. Their approach matches source to target by checking that they perform the same state transitions from lock to subsequent unlock operations and that the source does not allow more data-races. Morisset et al. [39] and Chakraborty and Vafeiadis [16] addressed this problem for the C/C++11 model, of which RA is a central fragment, by implementing matching algorithms between source and target that validate that all transformations between them have been independently proven to be safe under C/C++11.

Cho et al. [18] introduced a specialized semantics for *sequential* programs that can be used for justifying compiler optimizations under weak memory concurrency. They showed that behavior refinement under their sequential semantics implies refinement under any (sequential or parallel) context in the Promising Semantics 2.1 [17]. Their work focuses on optimizations of race-free accesses that are similar to C11's "non-atomics" [4, 32]. It cannot be used to establish the soundness of program transformations that we study in this paper. Adding non-atomics to our model is an important future work.

Denotational approaches were developed for models much weaker than RA [15, 24, 26, 28, 41] that allow the infamous Read-Write Reorder and thus, for a high-level programming language, require addressing the challenge of detecting semantic dependencies between instructions [3]. These approaches are based on summarizing multiple partial orders between actions that may arise when a given program is executed under some context. In contrast, we use totally ordered traces by relating to RA's interleaving operational semantics. In particular, Kavanagh and Brookes [28] use partial orders, Castellan, Paviotti et al. [15, 41] use event structures, and Jagadeesan et al., Jeffrey et al. [24, 26] employ "Pomsets with Preconditions" which trades compositionality for supporting non-multi-copy-atomicity, as in RA. These approaches do not validate certain access eliminations, nor Irrelevant Load Introduction, which our model validates.

An exciting aspect of our work is the connection between memory models to Moggi's monadic approach. For SC, Abadi and Plotkin, Dvir et al. [1, 20] have made an even stronger connection via algebraic theories [42]. These allow to modularly combine shared memory concurrency with other computational effects. Birkedal et al. [11] develop semantics for a type-and-effect system for SC memory which they use to enhance compiler optimizations based on assumptions on the context that come from the type system. We hope to the current work can serve as a basis to extend such accounts to weaker models.

Acknowledgments. Supported by the Israel Science Foundation (grant number 814/22) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811); and by a Royal Society University Research Fellowship and Enhancement Award.

References

1. Abadi, M., Plotkin, G.: A model of cooperative threads. Log. Methods Comput. Sci. **6**(4) (2010), `https://doi.org/10.2168/LMCS-6(4:2)2010`
2. Aguirre, A., Katsumata, S.y., Kura, S.: Weakest preconditions in fibrations. Mathematical Structures in Computer Science **32**(4) (2022), `https://doi.org/10.1017/S0960129522000330`
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: ESOP, LNCS, vol. 9032, Springer (2015), `https://doi.org/10.1007/978-3-662-46669-8_12`
4. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL, ACM (2011), `https://doi.org/10.1145/1926385.1926394`
5. Benton, N., Hofmann, M., Nigam, V.: Abstract effects and proof-relevant logical relations. In: POPL, ACM (2014)
6. Benton, N., Hofmann, M., Nigam, V.: Effect-dependent transformations for concurrent programs. In: PPDP, ACM (2016), `https://doi.org/10.1145/2967973.2968602`
7. Benton, N., Hughes, J., Moggi, E.: Monads and effects. In: APPSEM (2000)
8. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: PPDP, ACM (2007)
9. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations: higher-order store. In: PPDP, ACM (2009)
10. Benton, N., Leperchey, B.: Relational reasoning in a nominal semantics for storage. In: TLCA, Springer (2005)
11. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL, LIPIcs, vol. 16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012), `https://doi.org/10.4230/LIPIcs.CSL.2012.107`
12. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3) (2007), `https://doi.org/10.1016/j.tcs.2006.12.034`
13. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2) (1996), `https://doi.org/10.1006/inco.1996.0056`
14. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: CC, LNCS, vol. 6011, Springer (2010), `https://doi.org/10.1007/978-3-642-11970-5_7`
15. Castellan, S.: Weak memory models using event structures. In: JFLA, Saint-Malo, France (Jan 2016), URL `https://hal.inria.fr/hal-01333582`
16. Chakraborty, S., Vafeiadis, V.: Validating optimizations of concurrent C/C++ programs. In: CGO, ACM (2016), `https://doi.org/10.1145/2854038.2854051`
17. Cho, M., Lee, S., Hur, C., Lahav, O.: Modular data-race-freedom guarantees in the promising semantics. In: PLDI, ACM (2021), `https://doi.org/10.1145/3453483.3454082`

18. Cho, M., Lee, S., Lee, D., Hur, C., Lahav, O.: Sequential reasoning for optimizing compilers under weak memory concurrency. In: PLDI, ACM (2022), https://doi.org/10.1145/3519939.3523718
19. Dodds, M., Batty, M., Gotsman, A.: Compositional verification of compiler optimisations on relaxed memory. In: ESOP, LNCS, vol. 10801, Springer (2018), https://doi.org/10.1007/978-3-319-89884-1_36
20. Dvir, Y., Kammar, O., Lahav, O.: An algebraic theory for shared-state concurrency. In: APLAS, LNCS, vol. 13658, Springer (2022), https://doi.org/10.1007/978-3-031-21037-2_1
21. Dvir, Y., Kammar, O., Lahav, O.: A denotational approach to release/acquire concurrency (2024), URL http://tiny.cc/esop24ra
22. Hoare, T., van Staden, S.: The laws of programming unify process calculi. Sci. Comput. Program. **85** (2014), https://doi.org/10.1016/j.scico.2013.08.012
23. Hofmann, M.: Correctness of effect-based program transformations. In: Formal Logical Methods for System Security and Correctness, IOS Press (2008)
24. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with preconditions: a simple model of relaxed memory. Proc. ACM Program. Lang. **4**(OOPSLA) (2020), https://doi.org/10.1145/3428262
25. Jagadeesan, R., Petri, G., Riely, J.: Brookes is relaxed, almost! In: FOSSACS, LNCS, vol. 7213, Springer (2012), https://doi.org/10.1007/978-3-642-28729-9_12
26. Jeffrey, A., Riely, J., Batty, M., Cooksey, S., Kaysin, I., Podkopaev, A.: The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. Proc. ACM Program. Lang. **6**(POPL) (2022), https://doi.org/10.1145/3498716
27. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL, ACM (2017), https://doi.org/10.1145/3009837.3009850
28. Kavanagh, R., Brookes, S.: A denotational semantics for SPARC TSO. In: MFPS, ENTCS, vol. 341, Elsevier (2018), https://doi.org/10.1016/j.entcs.2018.03.025
29. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2) (Apr 2019), https://doi.org/10.1145/3326938.3326942
30. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL, ACM (2016), https://doi.org/10.1145/2837614.2837643
31. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM, LNCS, vol. 9995 (2016), https://doi.org/10.1007/978-3-319-48989-6_29
32. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, ACM (2017), https://doi.org/10.1145/3062341.3062352
33. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9) (1979), https://doi.org/10.1109/TC.1979.1675439

34. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL, ACM (2012), https://doi.org/10.1145/2103656.2103711
35. Liang, H., Feng, X., Fu, M.: Rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans. Program. Lang. Syst. **36**(1) (2014), https://doi.org/10.1145/2576235
36. Maillard, K., Hriţcu, C., Rivas, E., Van Muylder, A.: The next 700 relational program logics. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019), https://doi.org/10.1145/3371072
37. Manson, J., Pugh, W.W., Adve, S.V.: The Java memory model. In: POPL, ACM (2005), https://doi.org/10.1145/1040305.1040336
38. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1) (1991), https://doi.org/10.1016/0890-5401(91)90052-4
39. Morisset, R., Pawan, P., Nardelli, F.Z.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In: PLDI, ACM (2013), https://doi.org/10.1145/2491956.2491967
40. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs, LNCS, vol. 5674, Springer (2009), https://doi.org/10.1007/978-3-642-03359-9_27
41. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: ESOP, LNCS, vol. 12075, Springer (2020), https://doi.org/10.1007/978-3-030-44914-8_22
42. Plotkin, G., Power, J.: Notions of computation determine monads. In: FOSSACS, Springer Berlin Heidelberg (2002)
43. Poetzl, D., Kroening, D.: Formalizing and checking thread refinement for data-race-free execution models. In: TACAS, LNCS, vol. 9636, Springer (2016), https://doi.org/10.1007/978-3-662-49674-9_30
44. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. Proc. ACM Program. Lang. **2**(POPL) (2018), https://doi.org/10.1145/3158107
45. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI, ACM (2011), https://doi.org/10.1145/1993498.1993520
46. Turon, A.J., Wand, M.: A separation logic for refining concurrent objects. In: POPL, ACM (2011), https://doi.org/10.1145/1926385.1926415
47. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Nardelli, F.Z.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL, ACM (2015), https://doi.org/10.1145/2676726.2676995
48. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2) (1997), https://doi.org/10.1007/BF01211617

Intel PMDK Transactions: Specification, Validation and Concurrency*

Azalea Raad[1], Ori Lahav[2], John Wickerson[1], Piotr Balcer[3] and Brijesh Dongol[4](✉)

[1] Imperial College London, London, UK
[2] Tel Aviv University, Tel Aviv, Israel
[3] Intel, Gdansk, Poland
[4] University of Surrey, Guildford, UK

b.dongol@surrey.ac.uk

Abstract. Software Transactional Memory (STM) is an extensively studied paradigm that provides an easy-to-use mechanism for thread safety and concurrency control. With the recent advent of byte-addressable persistent memory, a natural question to ask is whether STM systems can be adapted to support *failure atomicity*. In this paper, we answer this question by showing how STM can be easily integrated with Intel's Persistent Memory Development Kit (PMDK) transactional library (which we refer to as txPMDK) to obtain STM systems that are both concurrent and persistent. We demonstrate this approach using known STM systems, TML and NOrec, which when combined with txPMDK result in persistent STM systems, referred to as PMDK-TML and PMDK-NORec, respectively. However, it turns out that existing correctness criteria are insufficient for specifying the behaviour of txPMDK and our concurrent extensions. We therefore develop a new correctness criterion, *dynamic durable opacity*, that extends the previously defined notion of durable opacity with dynamic memory allocation. We provide a model of txPMDK, then show that this model satisfies dynamic durable opacity. Moreover, dynamic durable opacity supports concurrent transactions, thus we also use it to show correctness of both PMDK-TML and PMDK-NORec.

1 Introduction

Persistent memory technologies (aka non-volatile memory, NVM) such as Memory-Semantic SSD [53] and XL-FLASH [13], combine the durability of hard drives with the fast and fine-grained accesses of DRAMs, with the potential to radically change how we build fault-tolerant systems. However, NVM also raises fundamental questions about semantics and the applicability of standard programming models.

* Raad is funded by a UKRI fellowship MR/V024299/1, EPSRC grant EP/X037029/1 and VeTSS. Lahav is supported by the Israel Science Foundation (grant 814/22) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811). Wickerson is funded by EPSRC grant EP/R006865/1. Dongol is funded by EPSRC grants EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, EP/R025134/2 and VeTSS.

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 150–179, 2024.
https://doi.org/10.1007/978-3-031-57267-8_6

```
struct loc {
  pmem::obj::p<int> value;
  pmem::obj::persistent_ptr<loc> next; };

struct root { pmem::obj::persistent_ptr<loc> head = nullptr; };

void post_crash(...) {
 auto pop = pmem::obj::pool<root>::open("file",...);
 auto root = pop.root();
 pmem::obj::transaction::run(pop, [&]{
    auto xvalue = root->head->value;
 }); }

int main(...) {
  auto pop = pmem::obj::pool<root>::open("file",...);
  auto root = pop.root();
  pmem::obj::transaction::run(pop, [&]{
    auto x = pmem::obj::make_persistent<loc>();
    x->value = 42;
    x->next = nullptr;
    root->head = x;
  }); }
```

Fig. 1: C++ snippet for allocating in persistent memory using TXPMDK [54]

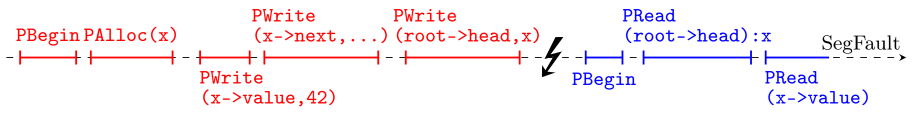
Among the most widely used collections of libraries for persistent programming is Intel's Persistent Memory Development Kit (PMDK), which was first released in 2015 [30]. One important component of PMDK is its transactional library, which we refer to as TXPMDK, and which supports generic *failure-atomic* programming. A programmer can use TXPMDK to protect against full system crashes by starting a transaction, performing transactional reads and writes, then committing the transaction. If a crash occurs during a transaction, but before the commit, then upon recovery, any writes performed by the transaction will be rolled back. If a crash occurs during the commit, the transaction will either be rolled back or be committed successfully, depending on how much of the commit operation has been executed. If a crash occurs after committing, the effect of the transaction is guaranteed to persist.

Most software transactional memory (STM) algorithms leave memory allocation implicit, since they are generally safe under standard allocation techniques (e.g. `malloc`). Memory that is allocated as part of a transaction can be deallocated if the transaction is aborted. However, in the context of persistency, memory allocation is more subtle since transactions may be interrupted by a crash.

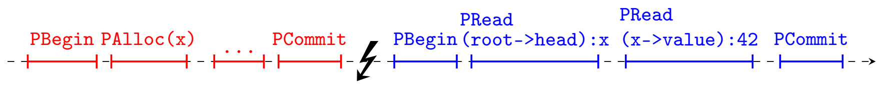
For example, consider the program in Fig. 1. Persistent memory is allocated, accessed and maintained via *memory pools* [54] (files that are memory mapped into the process address space) of a certain type (e.g. of type `loc` in Fig. 1). Due to address space layout randomization (ASLR) in most operating systems, the location of the pool can differ between executions and across crashes. As such, every pool has a root object from which all other objects in the pool can be found. That is, to avoid memory leaks, all objects in the pool must be reachable from the root. An application locates the root object using a *pool object pointer* (POP) that is to be created with every program invocation (e.g. line 15). After locating the pool root (line 16), we use a TXPMDK transaction (lines 17–22) to allocate a persistent `loc` object `x` (line 18) with value 42 (line 19) and add it to the pool (line 21).

Consider the scenario where the execution of this transaction crashes. After recovery from the crash, we then execute `post_crash` (line 7). As before, we open the pool (line 8) and locate its root (line 9). We then use a TXPMDK transaction to read from the `loc` object allocated and added at the pool head prior to the crash (line 11). There are then three cases to consider: the crash may have occurred **(1)** before the transaction started the commit process, **(2)** after the transaction successfully committed, or **(3)** while the transaction was in the process of committing.

In case **(1)**, the execution of the two transactions can be depicted as follows, where the `PBegin` events capture commencing the transactions (lines 17 and 10), `PAlloc(x)` denotes the persistent allocation of `x` (line 18); `PWrite(x->value,42)` captures writing to `x` (line 19); and `PRead(root->head):x` denotes reading from `x->value` and returning the value `x` (first part of line 11). As the first transaction never reached the commit stage, its effects (i.e. allocating `x` and writing to it) should be invisible (i.e. rolled back), and thus the read of the second transaction effectively reads from unallocated memory, leading to an error such as a segmentation fault.



In case **(2)**, the execution of the transactions is as follows, where the `PCommit` events capture the end (successful commit) of the transactions (lines 22 and 12), the effects of the first transaction fully persist upon successful commit, and thus the read in the second transaction does not fault.



Finally, in case **(3)**, either of the two behaviours depicted above is possible (i.e. the second transaction may either cause a segmentation fault or read from `x`).

Efficient and correct memory allocation in a persistent memory setting is challenging ([54, Chapter 16] and [55]). In addition to the ASLR issue mentioned above, the allocator must guarantee failure atomicity of heap operations on several internal data structures managed by PMDK. Therefore, PMDK provides its own allocator that is designed specifically to work with TXPMDK.

We identify two key drawbacks of TXPMDK as follows. In this paper, we take steps towards addressing both of these drawbacks.

A) Lack of concurrency support. Unlike existing STM systems in the persistent setting [39,32] that provide *both failure atomicity* (ensuring that a transaction either commits fully or not at all in case of a crash) *and isolation* (as defined by ACID properties, ensuring that the effects of incomplete transactions are invisible to concurrently executing transactions), TXPMDK only provides failure atomicity and does not offer isolation in concurrent settings. In particular, naïvely implemented applications with racy PMDK transactions lead to memory inconsistencies. This is against the spirit of STM: the primary function of STM systems is providing a concurrency control mechanism that ensures isolation. The current TXPMDK implementation provides two solutions: threads either execute *concurrent* transactions over *disjoint* parts of the memory [54, Chapter 7], or use user-defined *fine-grained locks* within a transaction to ensure memory isolation [54, Chapter 14]. However, both solutions are sub-optimal: the former enforces serial execution when transactions operate over the same part of the memory, and the latter expects too much of the user.

B) Lack of a suitable correctness criterion. There is no formal specification describing the desired behaviour of TXPMDK, and hence no rigorous description or correctness proof of its implementation. This undermines the utility of TXPMDK in safety-critical settings and makes it impossible to develop formally verified applications that use TXPMDK. Indeed, there is currently no correctness criterion for STM systems that provide dynamic memory allocation (a large category that includes all realistic implementations).

1.1 Concurrency for TXPMDK

Integrating concurrency with PMDK transactions is an important end goal for PMDK developers. The existing approach requires integration of locks with TXPMDK, which introduces overhead for programmers. Our paper shows that STM and PMDK can be easily combined, improving programmability. Many other works have aimed to develop failure-atomic and concurrent transactions (e.g. OneFile [52] and Romulus [16]), but none use off-the-shelf commercially available libraries. Moreover, these other works have not addressed correctness with the level of rigour that our paper does. In other work, popular key-value stores Memcached and Redis have been ported to use PMDK [36,37]; our work paves the way for concurrent version of these applications to be developed. Another example is the work of Chajed et al [11], who provide a simulation-based technique for *verifying* refinement of durable filesystems, where concurrency is handled by durable transactions.

We tackle the first drawback (A) mentioned above by developing, specifying, and validating two thread-safe versions of TXPMDK.

Contribution A: Making TXPMDK thread-safe. We combine TXPMDK with two off-the-shelf (thread-safe) STM systems, TML [17] and NOREC [18], to obtain two new implementations, PMDK-TML and PMDK-NOREC, that support *concurrent* failure-atomic transactions with dynamic memory allocation. In particular, we reuse the existing concurrency control mechanisms provided by these STM systems to ensure atomicity of write-backs, thus obtaining memory isolation even in a multi-threaded setting. We show that it is possible to integrate

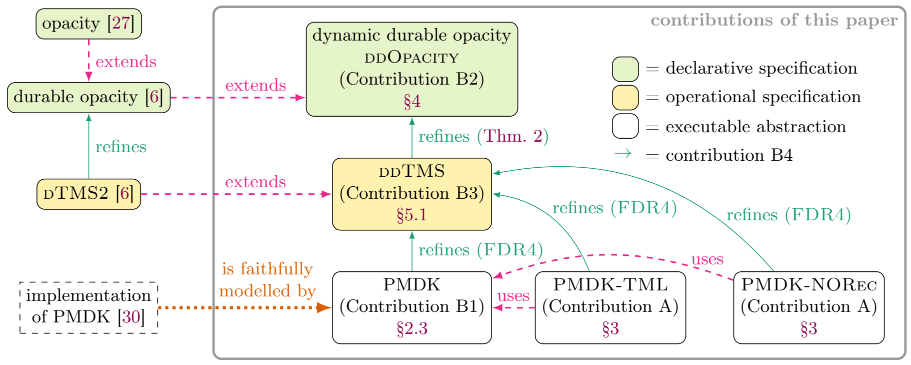

opacity [27]
extends
durable opacity [6]
refines
DTMS2 [6]
implementation of PMDK [30]
extends
extends
is faithfully modelled by
contributions of this paper
dynamic durable opacity ddOpacity (Contribution B2) §4
refines (Thm. 2)
ddTMS (Contribution B3) §5.1
refines (FDR4)
PMDK (Contribution B1) §2.3
uses
PMDK-TML (Contribution A) §3
uses
PMDK-NORec (Contribution A) §3
refines (FDR4)
refines (FDR4)
= declarative specification
= operational specification
= executable abstraction
= contribution B4

Fig. 2: The contributions of this paper and their relationships to prior work

these mechanisms with txPMDK to additionally achieve failure atomicity. Our approach is modular, with a clear separation of concerns between the isolation required due to concurrency and the atomicity required due to the possibility of system crashes. This shows that concurrency and failure atomicity are two orthogonal concerns, highlighting a pathway towards a mix-and-match approach to combining (concurrent) STM and failure-atomic transactions. Finally, in order to provide the same interface as PMDK, we extend both TML and NORec with an explicit operation for memory allocation.

1.2 Specification and Validation

To tackle drawback (B) above, we make four contributions. Together, they provide the first formal (and rigorous) specification of txPMDK and validation of its implementation.

Contribution B1: A model of txPMDK. We provide a formal specification of txPMDK as an abstract transition system. Our formal specification models almost all key components of txPMDK (including its redo and undo logs, as well as the interaction of these components with system crashes), with the exception of memory deallocation within txPMDK transactions.

Contribution B2: A correctness criterion for transactions with dynamic allocation. Although the literature includes several correctness criterion for transactional memory (TM), none can adequately capture txPMDK in that they do not account for dynamic memory allocation. We develop a new correctness condition, *dynamic durable opacity* (denoted ddOpacity), by extending durable opacity [6] to account for dynamic allocation. ddOpacity supports not only sequential transactions such as txPMDK, but also concurrent ones. To demonstrate the suitability of ddOpacity for concurrent and persistent (durable) transactions, later we validate our two concurrent txPMDK implementations (PMDK-NORec and PMDK-TML) against ddOpacity.

Contribution B3: An operational characterisation of our correctness criterion. Our aim is to show that txPMDK conforms to ddOpacity, or more precisely, that our model of TXPMDK refines our model of DDOPACITY. To demonstrate this, we use a new intermediate model called DDTMS. While DDOPACITY is defined declaratively, DDTMS is defined operationally, which makes it conceptually closer to our model of the TXPMDK implementation. We prove that DDTMS is a sound model of DDOPACITY (i.e. every trace of DDTMS satisfies DDOPACITY).

Contribution B4: Validation of TXPMDK, PMDK-TML and PMDK-NOREC in FDR4. We mechanise our implementations (TXPMDK, PMDK-TML and PMDK-NOREC) and specification (DDTMS) using the CSP modelling language. We use the FDR4 model checker [26] to show the implementations are refinements of DDTMS over both the persistent SC (PSC) [31] and persistent TSO ($Px86_{sim}$) [50] memory models. For $Px86_{sim}$, we use an equivalent formulation called $PTSO_{syn}$ developed by Khyzha and Lahav [31]. The proof itself is fully automatic, requiring no user input outside of the encodings of the models themselves. Additionally, we develop a sequential lower bound (DDTMS-Seq), derived from DDTMS, and show that this lower bound refines TXPMDK (and hence that TXPMDK is not vacuously strong). Our approach is based on an earlier technique for proving durable opacity [23], but incorporates much more sophisticated examples and memory models.

Outline. Fig. 2 gives an overview of the different components that we have developed in this paper and their relationships to each other and to prior work. We structure our paper by presenting the components of Fig. 2 roughly from the bottom up. In §2, we present the abstract TXPMDK model, and in §3 we describe its integration with STM to provide concurrency support via PMDK-TML and PMDK-NOREC. In §4 we present DDOPACITY, in §5 we present DDTMS, and in §6 we describe our FDR4 encodings and bounded proofs of refinement.

Additional Material. We provide our FDR4 development as supplementary material [47]. The proofs of all theorems are given in an extended version [46].

2 Intel PMDK transactions

We describe the abstract interface TXPMDK provides to clients (§2.1), our assumptions about the memory model over which TXPMDK is run (§2.2) and the operations of TXPMDK (§2.3). We present our PMDK abstraction in §2.3.

2.1 PMDK Interface

PMDK provides an extensive suite of libraries for simplifying persistent programming. The PMDK transactional library (TXPMDK) has been designed to support failure-atomicity by providing operations for tracking memory locations that are to be made persistent, as well allocating and accessing (reading and writing) persistent memory within an atomic block.

In Fig. 3 we present an example client code that uses TXPMDK. The code (due to [54, p. 131]) implements the `push` operation for a persistent linked-list queue.

```
 1 struct queue_node {
 2     pmem::obj::p<int> value;
 3     pmem::obj::persistent_ptr<queue_node> next; };
 4
 5 struct queue { private:
 6     pmem::obj::persistent_ptr<queue_node> head = nullptr;
 7     pmem::obj::persistent_ptr<queue_node> tail = nullptr; };
 8
 9 void push(pmem::obj::pool_base &pmem_op, int value) {
10    pmem::obj::transaction::run(pmem_op, [&]{
11       auto node = pmem::obj::make_persistent<queue_node>();
12       node->value = value;
13       node->next = nullptr;
14       if (head == nullptr) {
15          head = tail = node;
16       } else {
17          tail->next = node;
18          tail = node; }
19     }); }
```

Fig. 3: C++ persistent `push` operation using TXPMDK ([54, p. 131])

The implementation wraps a typical (non-persistent) `push` operation within a transaction using a C++ lambda `[&]` expression (line 10). The transaction is invoked using `transaction::run`, which operates over the memory pool `pmem_op`. The node structure (lines 2 and 3), the queue structure (lines 6 and 7), and any new node declaration (line 11) are to be tracked by a PMDK transaction. Additionally, the `push` operation takes as input the *persistent memory object pool*, `pmem_op`, which is a memory pool on which the transaction is to be executed. This argument is needed because the application memory may map files from different file systems. On line 7 we use `make_persistent` to perform a transactional allocation on persistent memory that is linked to the object pool `pmem_op` (see [54] for details). The remainder of the operation (lines 12–18) corresponds to an implementation of a standard `push` operation with (transactional) reads and writes on the indicated locations. At line 19, the C++ lambda and the transaction is closed, signalling that the transaction should be committed.

If the system crashes while `push` is executing, but before line 19 is executed, then upon recovery, the entire `push` operation will be rolled back so that the effect of the incomplete operation is not observed, and the queue remains a valid linked list. After line 19, the corresponding transaction executes a commit operation. If the system crashes during commit, depending on how much of the commit operation has been executed, the `push` operation will either be rolled back, or committed successfully. Note that roll-back in all cases ensures that the allocation at line 11 is undone.

2.2 Memory Models

We consider the execution of our implementations over two different memory models: PSC and PTSO$_{\text{syn}}$ [31]. Both models include a `flush x` instruction to persist the contents of the given location `x` to memory. PTSO_{syn} aims for fidelity to the Intel x86 architecture. In a race-free setting (as is the case for single-threaded TXPMDK transactions) it is sound to use the simpler PSC model, though we conduct all of our experiments in both models.

PSC is a simple model that considers persistency effects and their interaction with sequential consistency. Writes are propagated directly to per-location persistence buffers, and are subsequently flushed to non-volatile memory, either due to a system action, or the execution of a `flush` instruction. A read from `x` first attempts to fetch its value from the persistence buffer and if this fails, fetches its value from non-volatile memory.

Under Intel-x86, the memory models are further complicated by the interaction between *total store ordering* (TSO) effects [40] and persistency. Due to the abstract nature of our models (see Fig. 4) it is sufficient for us to focus on the simpler Px86_{sim} model [50] since we do not use any of the advanced features [48,49,50]. We introduce a further simplification via PTSO_{syn} that is *observationally equivalent* to Px86_{sim} [31]. Unlike Px86_{sim}, which uses a single (global) persistence buffer, PTSO_{syn} uses per-location buffers simplifying the resulting FDR4 models (§6).

In PTSO_{syn}, writes are propagated from the store buffer in FIFO order to a per-location FIFO persistency buffer. Writes in the persistency buffer are later persisted to the non-volatile memory. A read from location `x` first attempts to fetch the latest write to `x` from the store buffer. If this fails (i.e. no writes to `x` exists in the store buffer), it attempts to fetch the latest write from the persistence buffer of `x`, and if this fails, it fetches the value of `x` from non-volatile memory.

2.3 PMDK Implementation

We present the pseudo-code of our TXPMDK abstraction in Fig. 4. We model all features of TXPMDK (including its redo and and undo logs as well as its recovery mechanism in case of a crash) except memory deallocation within a TXPMDK transaction. We use `mem` to model the memory, mapping each location (in `loc`) to a value-metadata pair. We model a value (in `val`) as an integers, and `metadata` as a boolean indicating whether the location is allocated. As we see below, the list of free (unallocated) locations, `freeList`, is calculated during recovery using `metadata`.

Each PMDK transaction maintains redo logs and an undo log. The redo logs record the locations allocated by the transaction so that if a crash occurs while committing, the allocated locations can be reallocated, allowing the transaction to commit upon recovery. Specifically, TXPMDK uses two *distinct* redo logs: `tRedo` and `pRedo`. Both are associated with fields `undoValid` (which is unset when the log is invalidated), `checksum` (used to indicate whether the log is valid), and `allocs` (which contains the set of locations allocated by the transaction). Note that TXPMDK explicitly sets and unsets `undoValid`, whereas `checksum` is calculated (e.g. at line 36) and may be invalidated by crashes corrupting a partially completed write. The undo log records the original (overwritten) value of each location written to by the transaction, and is consulted if the transaction is to be rolled back. We model it as a map from locations to values (of type `int`).

```
// Each location is persistent; there is no explicitly volatile memory.
mem : loc -> {
  val : int; // the contents of this location
  metadata : bool; } // false = not allocated, true = allocated
freeList : loc list // transient list of free locations

// Redo logs -- tRedo is transient; pRedo is persistent.
tRedo$_t$, pRedo$_t$ : {undoValid:bool;  checksum:int;  allocs:loc set;}
undo$_t$ : loc -> int  // undo log recording the original val of each loc
undoValid : bool  // undoValid global flag, initially true
```

```
PBegin$_t$ $\triangleq$
   tRedo$_t$ := (true, -1, {})
   pRedo$_t$ := (true, -1, {})
   undo$_t$ := {}
   undoValid$_t$ := true

PAlloc$_t$ $\triangleq$
   x$_t$ := freeList.take
   tRedo$_t$.allocs :=
     tRedo$_t$.allocs $\cup$ {x$_t$}
   return x$_t$

PRead$_t$(x) $\triangleq$
   return mem[x].val

PWrite$_t$(x,v) $\triangleq$
   if x $\notin$ dom(undo$_t$) then
      w$_t$ := mem[x].val
      undo$_t$ := undo$_t$ $\cup$ {x $\mapsto$ w$_t$}
      flush undo$_t$
   mem[x].val := v

PCommit$_t$ $\triangleq$
   persist_writes$_t$
   tRedo$_t$.undoValid := false
   tRedo$_t$.checksum :=
         calc_checksum(tRedo$_t$)
   pRedo$_t$ := tRedo$_t$
   flush pRedo$_t$
   apply_pRedo$_t$
   pRedo$_t$.checksum := -1
   flush pRedo$_t$.checksum
```

```
apply_pRedo$_t$ $\triangleq$
   foreach x $\in$ pRedo$_t$.allocs:
      mem[x].metadata := true
      flush mem[x].metadata
   if $\neg$pRedo$_t$.undoValid  then
      undoValid$_t$ := false
      flush undoValid$_t$

persist_writes$_t$ $\triangleq$
   foreach x $\in$ dom(undo$_t$): flush x

roll_back$_t$ $\triangleq$
   foreach (x $\mapsto$ v) $\in$ undo$_t$:
      mem[x].val := v
   persist_writes$_t$

PAbort$_t$ $\triangleq$
   roll_back$_t$
   undoValid$_t$ := false
   flush undoValid$_t$
   foreach x $\in$ tRedo$_t$.allocs:
      freeList.add(x)

PRecovery$_t$ $\triangleq$
   if calc_checksum(pRedo$_t$)
      = pRedo$_t$.checksum
   then apply_pRedo$_t$
   if undoValid$_t$ then
      roll_back$_t$
   foreach x $\in$ dom(mem):
      if $\neg$mem[x].metadata then
         freeList.add(x)
```

Fig. 4: PMDK global variables and pseudo-code

A separate variable `undoValid` (distinct from `undoValid` in `tRedo` and `pRedo`) is used to determine whether this undo log is valid.

Each component in Fig. 4 have both a volatile and persistent copy, although some components, e.g. `tRedo` and `freeList`, are *transient*, i.e. their persistent versions are never used. Likewise, the persistent redo log, `pRedo`, is only used in a persistent fashion and its volatile copy is never used.

We now describe the operations in Fig. 4. We assume the operations are executed by a transaction with id t. This id is not useful in the sequential setting in which TXPMDK is used; however, in our concurrent extension (§3) the transaction id is critical.

PBegin. The begin operation simply sets all local variables to their initial values.

PAlloc. Allocation chooses and removes a free location, say `x`, from the free list, adds `x` to the transient redo log (line 20) and returns `x`. Removing `x` from `freeList` ensures it is not allocated twice, while the transient redo log is used together with the persistent redo log to ensure allocated locations are properly reallocated upon a system crash.

When the transaction commits, the transient redo log is copied to the persistent one (line 37), and the effect of the persistent log is applied at line 39 via `apply_pRedo`. (Note that `apply_pRedo` is also called by `PRecovery` on line 68.) The behaviour of this call depends on how much of the in-flight transaction was executed before the crash leading to the recovery. If a crash occurred after the transaction executed (line 37) and the corresponding write persisted (either due to a system flush or the execution of line 38), then executing `apply_pRedo` via `PRecovery` has the same effect as the executing line 39, i.e. the effect of the redo log will be applied. This (persistently) sets the `metadata` field of each location in the redo log to indicate that it is allocated (lines 43–45), and then invalidates the undo log (lines 46–48) so that the transaction is not rolled back.

PRead. A read from `x` simply returns its in-memory value (line 24). Note that location `x` may not be allocated; TXPMDK delegates the responsibility of checking whether it is allocated to the client.

PWrite. A write to `x` first checks (line 27) if the current transaction has already written to `x` (via a previously executed `PWrite`). If not, it logs the current value by reading the in-memory value of `x` (line 28) and records it in the undo log (line 29). The updated undo log is then made persistent (line 30). Once the current value of `x` is backed up in the undo log (either by the current write or by the previous write to `x`), the value of `x` in memory is updated to the new value `v` (line 31). As with the read, location `x` may not have been allocated; TXPMDK delegates this check to the client.

PCommit. The main idea behind the commit operation is to ensure all writes are persisted, and that the persistent redo and undo logs are cleared in the correct order, as follows. **(1)** On line 34 all writes written by the transaction are persisted. **(2)** Next, the transient redo log is invalidated (line 35) and the checksum for the log is calculated (line 36). This updated transient log is then set to be the persistent redo log (line 37), which is then made persistent (line 38). Note that after executing line 38, we can be assured that the transaction has committed; if a crash occurs after this point, the recovery will *redo* and persist the allocation and the undo log will be cleared. **(3)** The operation then calls `apply_pRedo` at line 39, which makes the allocation persistent and clears the undo log. **(4)** Finally, at line 40, the `pRedo` checksum is invalidated since `apply_pRedo` has already been executed. If a crash occurs after line 40 has been executed, then the recovery checks at line 67 and line 69 will fail, i.e. recovery will calculate the free list.

PAbort. A PMDK transaction is aborted by a `PRead`/`PWrite` that attempts to access (read/write) an unallocated location. When a transaction is aborted, all of its observable effects must be rolled back. First, the memory effects are rolled back (line 59), then the undo log is invalidated (line 60) and made persistent (line 61), preventing undo from being replayed in case a crash occurs. Finally, all of the locations allocated by the executing transaction are freed (lines 62–63). Note that if a crash occurs during an abort, the effect of the abort will be replayed. `PRecovery` reconstructs the free list at lines 71–73, which effectively replays the loop at lines 62–63 of `PAbort`. Additionally, if a crash occurs before the write at line 60 has persisted, then the effect of undoing the operation will be explicitly replayed by the roll-back executed by `PRecovery` since `undoValid` holds. If the crash occurs after the write at line 60 has persisted, then no roll-back is necessary.

PRecovery. The recovery operation is executed immediately after a crash, and before any other operation is executed. The recovery proceeds in three phases: **(1)** The checksum of the persistent redo log is recalculated (line 67) and if it matches the stored checksum (`pRedo.checksum`) the `apply_pRedo` operation is executed. As discussed, `apply_pRedo` sets and persists the metadata of each location in the redo log, and then invalidates the undo log. **(2)** The transaction is rolled back if `apply_pRedo` in step 1 fails; otherwise, no roll-back is performed. **(3)** The free list is reconstructed by inserting each location whose metadata is set to false into `freeList` (lines 71–73).

Correctness and Thread Safety. As discussed in §2.1, TXPMDK is designed to be failure-atomic. This means that correctness criteria such as opacity [27,2] and TMS1/TMS2 [20] (restricted to sequential transactions) are inadequate since they do not accommodate crashes and recovery. This points to conditions such as durable opacity [6], which extends opacity with a persistency model. However, durable opacity (restricted to sequential transactions) is also insufficient since it does not define correctness of allocations and assumes totally ordered histories. In §4 we develop a generalisation of durable opacity, called *dynamic durable opacity* (DDOPACITY) that addresses both of these issues. As with durable opacity, DDOPACITY defines correctness for concurrent transactions. We develop concurrent extensions of PMDK transactions in §3, which we show to be correct against (i.e. refinements of) DDOPACITY.

As discussed, PMDK transactions are not thread-safe; e.g. concurrent calls to `PRead` and `PWrite` on the same location create a data race causing `PRead` to return an undefined value (see the example in §1). We discuss techniques for mitigating against such races in §3. Nevertheless, some PMDK transactional operations are naturally thread-safe. In particular, `PAlloc` is designed to be thread-safe via an built-in *arena* mechanism: a memory pool split into disjoint arenas with each thread allocating from its own arena. Moreover, each thread uses locks for each arena to publish allocated memory to the shared pool [55].

Init. glb = 0

```
$TxBegin_t \triangleq$
   do $loc_t$ := glb
   until even($loc_t$)
   $PBegin_t$

$TxAlloc_t \triangleq$
   return $PAlloc_t$

$TxWrite_t$(x, v) $\triangleq$
   if even($loc_t$) then
      if $\neg$cas(glb, $loc_t$, $loc_t$+1)
      then $PAbort_t$; return abort
      else $loc_t$++
   $PWrite_t$(x,v)
```

```
$TxRead_t$(x) $\triangleq$
  $v_t$ := $PRead_t$(x)
  if even($loc_t$) then
     if glb = $loc_t$ then
        return $v_t$
     else $PAbort_t$; return abort
  else return $v_t$

$TxCommit_t \triangleq$
   $PCommit_t$
   if odd($loc_t$) then
      glb := $loc_t$+1

Recovery $\triangleq$
   foreach $t \in$ TXID:
      $PRecovery_t$
   glb := 0
```

Fig. 5: Pseudo-code for PMDK-TML with our additions made w.r.t. TML highlighted red

3 Making PMDK Transactions Concurrent

We develop two algorithms that combine two existing STM systems with PMDK. The first algorithm (§3) is based on TML [17], which uses *pessimistic concurrency control* via an eager write-back scheme. Writing transactions effectively take a lock and perform the writes in place. The second algorithm (§3) is based on NOREC [18], which utilises *optimistic concurrency control* via a lazy write-back scheme. In particular, transactional writes are *collected* in a local write set and *written back* when the transaction commits.

It turns out that PMDK can be incorporated within both algorithms straightforwardly. This is a strength of our approach and points towards a generic technique for extending existing STM systems with failure atomicity. Given the challenges of persistent allocation, we reuse PMDK's allocation mechanisms to provide an explicit allocation mechanism in both our extensions [54].

PMDK-TML. We present the pseudo-code for PMDK-TML (combining TML and TXPMDK) in Fig. 5, where we highlight the calls to TXPMDK operations. These calls are the only changes we have made to the TML algorithm. TML is based on a single global counter, `glb`, whose value is read and stored within a local variable loc_t when transaction t begins (`TxBegin`). There is an in-flight *writing* transaction iff `glb` is odd. TML is designed for read-heavy workloads, and thus allows multiple concurrent read-only transactions. A writing transaction causes all other concurrent transactions to abort.

PMDK-TML proposes a modular combination of PMDK with the TML algorithm by nesting a PMDK transaction inside a TML transaction; i.e. each transaction additionally starts a PMDK transaction. All reads and writes to memory are replaced by txPMDK read and write operations. Moreover, when a transaction aborts or commits, the operation calls a txPMDK abort or commit, respectively. Finally, PMDK-TML includes allocation and recovery operations, which call txPMDK allocation and recovery, respectively. The recovery operation additionally sets `glb` to 0.

A read-only transaction t may call $\texttt{PRead}_t$ at line 16 when another transaction t' is executing $\texttt{PWrite}_{t'}$ at line 14 on the same location. Since txPMDK does not guarantee thread safety for these calls, the value returned by $\texttt{PRead}_t$ should not be passed back to the client. This is indeed what occurs. First, note that if transaction t is read-only, then $\texttt{loc}_t$ is even. Moreover, a read-only transaction only returns the value returned by $\texttt{PRead}_t$ (line 19) if no other transaction has acquired the lock since t executed $\texttt{TxBegin}_t$. In the scenario described above, t' must have incremented `glb` by successfully executing the CAS at line 11 as part of the first write operation executed by t', changing the value of `glb`. This means that t would abort since the test at line 18 would fail.

PMDK-NORec. We present PMDK-NORec (combining NOrec and PMDK) in Fig. 6, where we highlight the calls to txPMDK. These calls are the only changes we have made to the NOrec algorithm. As with TML, NOrec is based on a single global counter, `glb`, whose value is read and stored within a transaction-local variable `loc` when a transaction begins (`TxBegin`). There is an in-flight writing transaction iff `glb` is odd. Unlike TML, NOrec performs lazy write-back, and hence utilises transaction-local read and write sets. A transaction only performs the write-back at commit time once it "acquires" the `glb` lock. Prior to write-back and read response, it ensures that the read sets are consistent using a per-location validate operation. We eschew details of the NOrec synchronisation mechanisms and refer the interested reader to the original paper [18].

The transformation from txPMDK to PMDK-NORec is similar to PMDK-TML. We ensure that a PMDK transaction is started when a PMDK-NORec transaction begins, and that this PMDK transaction is either aborted or committed before the PMDK-NORec transaction completes. We introduce `TxAlloc` and `Recovery` operations that are identical to PMDK-TML, and replace all calls to read and write from memory by `PRead` and `PWrite` operations, respectively.

As with PMDK-TML, a `PRead` executed by a transaction (at line 12, line 15 or line 31) may race with a `PWrite` (at line 43) executed by another transaction. However, since `PWrite` operations are only executed after a transaction takes the `glb` lock (at line 40), any transaction with a racy `PRead` is revalidated. If validation fails, the associated transaction is aborted.

4 A Declarative Correctness Criteria

We present a declarative correctness criteria for TM implementations. Unlike prior definitions such as (durable) opacity, TMS1/2 etc. that are defined in terms of histories of invocations and responses, we define dynamic durable opacity (ddOpacity) in terms of execution graphs, as is standard model for weak memory setting. Our models are inspired by prior work on declarative specifications for

Init: `glb = 0`

```
1  TxBegin_t ≜
2     do loc_t := glb
3     until even(loc_t)
4     PBegin_t
5
6  TxAlloc_t ≜
7     return PAlloc_t
8
9  TxRead_t(x) ≜
10    if x ∈ dom(wrSet_t) then
11       return wrSet_t(x)
12    v_t := PRead_t(x)
13    while loc_t ≠ glb
14       loc_t := Validate
15       v_t := PRead_t(x)
16    rdSet_t := rdSet_t ∪ {x ↦ v_t}
17    return v_t
18
19 Recovery ≜
20    foreach t ∈ TXID:
21       PRecovery_t
22    glb := 0

23 TxWrite_t(x,v) ≜
24    wrSet_t := wrSet_t ∪ {x ↦ v}
25
26 Validate_t ≜
27    while true
28       time_t := glb
29       if odd(time_t) then goto 28
30       foreach x ↦ v ∈ rdSet_t:
31          if PRead_t(x) ≠ v
32          then PAbort_t; return abort
33       if time_t = glb
34       then return time_t
35
36 TxCommit_t ≜
37    if wrSet_t.isEmpty
38       then PCommit_t
39            return
40    while ¬cas(glb, loc_t, loc_t + 1)
41       loc_t := Validate_t
42    foreach x ↦ v ∈ wrSet_t:
43       PWrite_t(x, v)
44    PCommit_t
45    glb := loc_t + 2
46    return
```

Fig. 6: Pseudo-code for PMDK-NOREC, with our additions made w.r.t. NOREC highlighted red

transactional memory, which focussed on specifying relaxed transactions [22,14]. However, these prior works do not describe crashes or allocation.

Executions and Events. The traces of memory accesses generated by a program are commonly represented as a set of *executions*, where each execution G is a graph comprising: 1. a set of events (graph nodes); and 2. a number of relations on events (graph edges). Each event e corresponds to the execution of either a transactional event (e.g. marking the beginning of a transaction) or a memory access (read/write) within a transaction.

Definition 1 (Events). *An* event *is a tuple* $a = \langle n, \tau, t, l \rangle$, *where* $n \in \mathbb{N}$ *is an event identifier,* $\tau \in \textit{TID}$ *is a* thread identifier, $t \in \textit{TXID}$ *is a* transaction identifier *and* $l \in \textsc{Lab}$ *is an event* label.

A label may be `B` *to mark the beginning of a transaction;* `A` *to denote a transactional abort;* $(\texttt{M}, x, 0)$ *to denote a memory allocation yielding* x *initialised with value* 0; $(\texttt{R}, x, v)$ *to denote reading value* v *from location* x; $(\texttt{W}, x, v)$ *to denote writing* v *to* x; `C` *to mark the beginning of the transactional commit process; or* `S` *to denote a successful commit.*

The functions `tid`, `tx` and `lab` respectively project the thread identifier, transaction identifier and the label of an event. The functions `loc`, $\texttt{val}_\texttt{r}$ and $\texttt{val}_\texttt{w}$

respectively project the location, the read value and the written value of a label, where applicable, and are lifted to events by defining e.g. $\mathtt{loc}(a) = \mathtt{loc}(\mathtt{lab}(a))$.

Notation. Given a relation r and a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$; $\mathsf{irreflexive}(r)$ for $\nexists a.\, (a,a) \in r$; and $\mathsf{acyclic}(\mathsf{r})$ for $\mathsf{irreflexive}(\mathsf{r}^+)$. We write $\mathsf{r}_1; \mathsf{r}_2$ for the relational composition of r_1 and r_2, i.e. $\{(a,b) \mid \exists c.\, (a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$. When A is a set of events, we write A_x for $\{a \in A \mid \mathtt{loc}(a){=}x\}$, and r_x for $\mathsf{r}|_{A_x}$. Analogously, we write A_t for $\{a \in A \mid \mathtt{tx}(a){=}t\}$. The *'same-transaction' relation*, $\mathsf{st} \subseteq E \times E$, is the equivalence relation $\mathsf{st} \triangleq \left\{(a,b) \in E \times E \,\middle|\, \mathtt{tx}(a){=}\mathtt{tx}(b)\right\}$.

Definition 2. *An* execution*, $G \in \text{Exec}$, is a tuple* $(E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$*, where:*

- *E is a set of* events*. The set of* reads *in E is* $R \triangleq \left\{e \in E \,\middle|\, \mathtt{lab}(e){=}(\mathtt{R}, -, -)\right\}$*. The sets of* allocations *(M),* writes *(W),* aborts *(A),* transactional begins *(B),* transactional commits *(C) and* commit successes *(S) are analogous.*
- $\mathsf{po} \subseteq E \times E$ *denotes the* 'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread.*
- $\mathsf{clo} \subseteq E \times E$ *denotes the* 'client-order' relation*, which is a strict partial order between transactions* ($\mathsf{st}; \mathsf{clo}; \mathsf{st} \subseteq \mathsf{clo} \setminus \mathsf{st}$) *that extends the program order between transactions* ($\mathsf{po} \setminus \mathsf{st} \subseteq \mathsf{clo}$)*.*
- $\mathsf{rf} \subseteq (M \cup W) \times R$ *denotes the* 'reads-from' relation *between events of the same location with matching values; i.e.* $(a,b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a){=}\mathtt{loc}(b) \wedge \mathtt{val}_\mathtt{w}(a){=}\mathtt{val}_\mathtt{r}(b)$*. Moreover,* rf *is total and functional on its range.*
- $\mathsf{mo} \subseteq E \times E$ *is the* 'modification-order'*, defined as the disjoint union of relations* $\{\mathsf{mo}_x\}_{x \in \text{Loc}}$*, such that each* mo_x *is a strict total order on* $M_x \cup W_x$*.*

Given a relation $\mathsf{r} \subseteq E \times E$, we write r_T for lifting r to transaction classes: $\mathsf{r}_\mathsf{T} \triangleq \mathsf{st}; (\mathsf{r} \setminus \mathsf{st}); \mathsf{st}$. For instance, when $(w, r) \in \mathsf{rf}$, w is a transaction t_1 event and r is a transaction t_2 event, then all events in t_1 are rf_T-related to all events in t_2. We write r_I to restrict r to its *intra-transactional* edges (within a transaction): $\mathsf{r}_\mathsf{I} \triangleq \mathsf{r} \cap \mathsf{st}$; and write r_E to restrict r to its *extra-transactional* edges (outside a transaction): $\mathsf{r}_\mathsf{E} \triangleq \mathsf{r} \setminus \mathsf{st}$. Analogously, we write r_i to restrict r to its *intra-thread edges*: $\mathsf{r}_\mathsf{i} \triangleq \left\{(a,b) \in \mathsf{r} \,\middle|\, \mathtt{tid}(a){=}\mathtt{tid}(b)\right\}$; and write r_e to restrict r to its *extra-thread edges*: $\mathsf{r}_\mathsf{e} \triangleq \mathsf{r} \setminus \mathsf{r}_\mathsf{i}$.

In the context of an execution G (we use the "$G.$" prefix to make this explicit), the *reads-before* relation is $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo})$.

Lastly, we write Commit for the events of *committing* transactions, i.e. those that have reached the commit stage: $\mathsf{Commit} \triangleq \mathit{dom}(\mathsf{st}; [C])$. We define the sets of *aborted* events, Abort, and *(commit)-successful* events, Succ, analogously. We define the set of *commit-pending* events as $\mathsf{CPend} \triangleq \mathsf{Commit} \setminus (\mathsf{Abort} \cup \mathsf{Succ})$, and the set of *pending* events as $\mathsf{Pend} \triangleq E \setminus (\mathsf{CPend} \cup \mathsf{Abort} \cup \mathsf{Succ})$.

Given an execution $G{=}(E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$, we write $G|_A$ for $(E \cap A, \mathsf{po}|_{E \cap A}, \mathsf{clo}|_{E \cap A}, \mathsf{rf}|_{E \cap A}, \mathsf{mo}|_{E \cap A})$. We further impose certain "well-formedness" conditions on executions, used to delimit transactions and restrict allocations. For example, we require that events of the same transaction are by the same thread and the each t contains exactly one begin event. In particular, these conditions ensure that in the context of a well-formed execution G we have 1. $G.\mathsf{Succ} \subseteq G.\mathsf{Commit}$; 2. each t contains at most a single abort or success ($|G.E_t \cap (A \cup S)| \leq 1$) and thus $G.(\mathsf{Succ} \cap \mathsf{Abort}) = \emptyset$; and 3. $G.E = G.(\mathsf{Pend} \uplus \mathsf{Abort} \uplus \mathsf{CPend} \uplus \mathsf{Succ})$, i.e. the sets $G.\mathsf{Pend}$, $G.\mathsf{Abort}$, $G.\mathsf{CPend}$ and $G.\mathsf{Succ}$ are pair-wise disjoint.

Execution Consistency. The definition of (well-formed) executions above puts very few constraints on the rf and mo relations. Such restrictions and thus the permitted behaviours of a transactional program are determined by defining the set of *consistent* executions, defined separately for each transactional consistency model. The existing literature includes several definitions of well-known consistency models, including *serialisability* (SER) [41], *snapshot isolation* (SI) [9,44] and *parallel snapshot isolation* (PSI) [10,43].

Serialisability (SER). The *serialisability* (SER) consistency model [41] is one of the most well-known transactional consistency models, as it provides strong guarantees that are intuitive to understand and reason about. Specifically, under SER, all concurrent transactions must appear to execute atomically one after another in a *total sequential order*. The existing declarative definitions of SER [9,10,50] are somewhat restrictive in that they only account for fully committed (complete) transactions, i.e. they do not support pending or aborted transactions. Under the assumption that all transactions are complete, an execution $(E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$ is deemed to be serialisable (i.e. SER-consistent) iff:

– $\mathsf{rf}_\mathrm{I} \cup \mathsf{mo}_\mathrm{I} \cup \mathsf{rb}_\mathrm{I} \subseteq \mathsf{po}$ (SER-INT)
– $\mathsf{clo} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$ is acyclic. (SER-EXT)

The SER-INT axiom enforces *intra-transactional* consistency, ensuring that e.g. a transaction observes its own writes by requiring $\mathsf{rf}_\mathrm{I} \subseteq \mathsf{po}$ (i.e. intra-transactional reads respect the program order). Analogously, the SER-EXT axiom guarantees *extra-transactional* consistency, ensuring the existence of a total sequential order in which all concurrent transactions appear to execute atomically one after another. This total order is obtained by an arbitrary extension of the (partial) 'happens-before' relation which captures synchronisation resulting from transactional orderings imposed by client order (clo) or *conflict* between transactions ($\mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$). Two transactions are conflicted if they both access (read or write) the same location x, and at least one of these accesses is a write. As such, the inclusion of $\mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T}$ enforces conflict-freedom of serialisable transactions. For instance, if transactions t_1 and t_2 both write to x via events w_1 and w_2 such that $(w_1, w_2) \in \mathsf{mo}$, then t_1 must commit before t_2, and thus the entire effect of t_1 must be visible to t_2.

Opacity. We do not stipulate that all transactions commit successfully and allow for both aborted and pending transactions. As such, we opt for the stronger notion of transactional correctness known as *opacity*. In what follows we describe our notion of opacity over executions (formalised in Def. 3), and later relate it to the existing notion of opacity over histories [27] and prove that our characterisation of opacity is *equivalent* to that of the existing one (see Thm. 1). Further intuitions are provided in the extended version of this paper [46].

Definition 3 (Opacity). *An execution* $G = (E, \mathsf{po}, \mathsf{clo}, \mathsf{rf}, \mathsf{mo})$ *is* opaque *iff:*

- $dom(\mathsf{rf}_\mathsf{T}) \subseteq \mathsf{Vis}$ *(VIS-RF)*
- $\mathsf{rf}_\mathrm{I} \cup \mathsf{mo}_\mathrm{I} \cup \mathsf{rb}_\mathrm{I} \subseteq \mathsf{po}$ *(INT)*
- $(\mathsf{clo} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup (\mathsf{rb}_\mathsf{T}; [\mathsf{Vis}]))$ *is acyclic* *(EXT)*

where $\mathsf{Vis} \triangleq \mathsf{Succ} \cup \mathsf{CPendRF}$ *with* $\mathsf{CPendRF} \triangleq dom([\mathsf{CPend}]; \mathsf{rf}_\mathsf{T})$.

The existing definition of opacity [27] does not account for memory allocation and assumes that all locations accessed (read/written) by a transaction are initialised with some value (typically 0). In our setting, we make no such assumption and extend the notion of opacity to *dynamic opacity* to account for memory allocation. More concretely, our goal is to ensure that accesses in *visible* transactions are *valid*, in that they are on locations that have been previously allocated in a visible transaction. We define an execution to be dynamically opaque (Def. 4) if its visible write accesses are valid, i.e. are mo-preceded by a visible allocation.

Definition 4 (Dynamic opacity). *An execution G is* dynamically opaque *iff it is opaque (Def. 3) and* $G.(W \cap \mathsf{Vis}) \subseteq rng\big([M \cap \mathsf{Vis}]; G.\mathsf{mo}\big)$.

We next use the above definitions to define (dynamic durable) opacity over execution *histories*. In the context of persistent memory where executions may crash (e.g. due to a power failure) and resume thereafter upon recovery, a history is a sequence of events (Def. 1) partitioned into different *eras* separated by *crash markers* (recording a crash occurrence), provided that the threads in each era are distinct, i.e. thread identifiers from previous eras are not reused after a crash.

Definition 5 (Histories). *A* history*, $H \in \textsc{Hist}$, is a pair (E, to), where E comprises events and* crash markers*, $E \subseteq \textsc{Event} \cup \textsc{Crash}$ with $\textsc{Crash} \triangleq \big\{(n, ⚡) \,\big|\, n \in \mathbb{N}\big\}$, and to is a total order on E, such that:*

- $(E, \mathsf{to}_\mathsf{i})$ *is well-formed; and*
- *events separated by crashes have distinct threads:*
  $([E]; \mathsf{to}; [\textsc{Crash}]; \mathsf{to}; [E]) \cap \mathsf{to}_\mathsf{i} = \emptyset$.

A history (E', pto) is a prefix *of history (E, to) iff $E' \subseteq E$, $\mathsf{pto} = \mathsf{to}|_{E'}$ and $dom(\mathsf{to}; [E']) \subseteq E'$.*

The *client order* induced by a history $H = (E, \mathsf{to})$, denoted by $\mathsf{clo}(H)$, is the partial order on TXID defined by $\mathsf{clo}(H) \triangleq [S \cup A]; \mathsf{to}_\mathsf{T}; [B]$. We define history opacity as a *prefix-closed* property (*cf.* [27]), designating a history H as opaque if *every prefix* (E, pto) of H induces an opaque execution. The notion of dynamic opacity over histories is defined analogously.

Definition 6. *A history H is* opaque *iff for each prefix $H_p = (E, \mathsf{pto})$ of H, there exist rf, mo such that $(E, \mathsf{pto}_\mathsf{i}, \mathsf{clo}(H_p), \mathsf{rf}, \mathsf{mo})$ is opaque (Def. 3). H is* dynamically opaque *iff for each prefix $H_p{=}(E, \mathsf{pto})$ of H, there exist rf, mo such that $(E, \mathsf{pto}_\mathsf{i}, \mathsf{clo}(H_p), \mathsf{rf}, \mathsf{mo})$ is dynamically opaque (Def. 4).*

We define *durable opacity* over histories: a history H is durably opaque iff the history obtained from H by removing crash markers is opaque. We define *dynamic, durable opacity* analogously.

Definition 7. *A history* (E, to) *is* durably opaque *iff* $(E \setminus \textsc{Crash}, \mathsf{to}|_{E\setminus\textsc{Crash}})$ *is opaque. A history* (E, to) *is* dynamically and durably opaque *iff the history* $(E \setminus \textsc{Crash}, \mathsf{to}|_{E\setminus\textsc{Crash}})$ *is dynamically opaque.*

Finally, we show that our definitions of history (durable) opacity are equivalent to the original definitions in the literature. (See [46] for the proof.)

Theorem 1. *History opacity as defined in Def. 6 is equivalent to the original notion of opacity [27]. History durable opacity as defined in Def. 7 is equivalent to the original notion of durable opacity [6].*

5 Operationally Proving Dynamic Durable Opacity

We develop an operational specification, ddTMS (§5.1), and prove it correct against ddOpacity (§5.2). In particular, we show that every history (i.e. observable trace) of ddTMS satisfies ddOpacity. As ddTMS is a concurrent operational specification, it serves as basis for validating the correctness of txPMDK as well as our concurrent extensions PMDK-TML and PMDK-NORec.

5.1 ddTMS: The dTMS2 Automaton Extended with Allocation

ddTMS is based on dTMS2, which is an operational specification that guarantees durable opacity [6]. dTMS2 in turn is based on TMS2 automaton [20], which is known to satisfy opacity [33]. Furthermore, the ddTMS commit operation includes the simplification described by Armstrong et al [1], omitting a validity check when committing read-only transactions. In what follows we present ddTMS as a transition system.

ddTMS state. Formally, the state of ddTMS is given by the variables in Fig. 7. dTMS2 keeps track of a sequence of memory stores, mems, one for each committed writing transaction since the last crash. This allows us to determine whether reads are consistent with previously committed write operations. Each committing transaction that contains at least one write adds a new memory version to the end of the memory sequence. As we shall see, mems tracks allocated locations since it maps every allocated location to a value different from $\perp$.

Each transaction t is associated with several variables: pc_t, $beginIdx_t$, $rdSet_t$, $wrSet_t$ and $alSet_t$. The pc_t denotes the program counter, ranging over a set of *program counter values* ensuring each transaction is well-formed and that each transactional operation takes effect between its invocation and response. The $beginIdx_t \in \mathbb{N}$ denotes the *begin index*, set to the index of the most recent memory version when the transaction begins. This is used to ensure the real-time ordering property between transactions. The $rdSet_t \in \textsc{Loc} \rightharpoonup \textsc{Val}$ is the *read set* and $wrSet_t \in \textsc{Loc} \rightharpoonup \textsc{Val}$ is the *write set*, recording the values read and written by

$$
\begin{aligned}
\mathsf{mems} \in \textsc{Mem} &\triangleq \textsc{Seq}\,\langle \textsc{Loc} \to \textsc{Val}_\perp \rangle \qquad \textsc{Val}_\perp \triangleq \textsc{Val} \cup \{\perp\}, \text{where } \perp \notin \textsc{Val} \\
\mathsf{S} \in \textsc{State} &\triangleq \textsc{TXId} \to \textsc{TState} \\
\mathsf{s} \in \textsc{TState} &\triangleq \mathbb{N} \times (\textsc{Loc} \rightharpoonup \textsc{Val}) \times (\textsc{Loc} \rightharpoonup \textsc{Val}) \times \mathcal{P}(\textsc{Loc}) \\
&\quad \text{storing the local begin index, read set, write set and allocation set} \\
\mathsf{PC} \in \textsc{PCMap} &\triangleq \textsc{TXId} \to \textsc{PCVal} \\
\textsc{Invs} &\triangleq \left\{ \begin{matrix} \texttt{TxBegin}, \texttt{TxRead}(l), \texttt{TxWrite}(l,v), \\ \texttt{TxAlloc}, \texttt{TxCommit} \end{matrix} \,\middle|\, l \in \textsc{Loc}, v \in \textsc{Val} \right\} \\
\textsc{Resps} &\triangleq \left\{ \begin{matrix} \texttt{TxBegin}, \texttt{TxRead}(l,v), \texttt{TxWrite}(l,v), \\ \texttt{TxAlloc}(l), \texttt{TxCommit}, \texttt{Abort} \end{matrix} \,\middle|\, l \in \textsc{Loc}, v \in \textsc{Val} \right\} \\
\textsc{PCVal} &\triangleq \left\{ \text{init}, \text{ready}, \text{aborted}, \text{committed}, \text{fault}, \mathit{\Pi}(i), \mathit{\Delta}(\texttt{TxCommit}) \,\middle|\, i \in \textsc{Invs} \right\} \\
\alpha \in \textsc{Action} &\triangleq \left\{ \mathit{inv}(i), \mathit{res}(r), \varepsilon, \lightning \,\middle|\, i \in \textsc{Invs}, r \in \textsc{Resps} \right\}
\end{aligned}
$$

Initially, $\mathsf{PC}_0 \triangleq \lambda t.\text{init} \qquad \mathsf{S}_0 \triangleq \lambda t.(0, \emptyset, \emptyset, \emptyset) \qquad \mathsf{mems}_0 \triangleq [\lambda x.\ \perp]$

Fig. 7: DDTMS state

the transaction during its execution, respectively. We use $S \rightharpoonup T$ to denote a partial function from S to T. Finally, $\mathit{alSet}_t \subseteq \textsc{Loc}$ denotes the *allocation set*, containing the set of locations allocated by the transaction t. We use s.beginIdx, s.rdSet, s.wrSet and s.alSet to refer to the begin index, read set, write set and allocation set of a state s, respectively.

The read set is used to determine whether the values read by the transaction are consistent with its version of memory (using validIdx). The write set, on the other hand, is required because writes are modelled using *deferred update* semantics: writes are recorded in the transaction's write set and are not published to any shared state until the transaction commits.

DDTMS Global Transitions. DDTMS is specified by the transition system shown in Fig. 8, where the DDTMS global transitions are given at the top and the per-transaction transitions are given at the bottom. The global transitions may either take a per-transaction step (rule (S)), match a transaction fault (rule (F)), crash (rule (X)), or behave chaotically due to a fault (rule (C)).

Note that a *crash* transition models both a crash and a recovery. It sets the program counter of every live transaction to aborted, preventing them from performing any further actions after the crash. Since transaction identifiers are not reused, the program counters of completed transactions need not be modified. After restarting, it must not be possible for any new transaction to interact with stale memory states prior to the crash. Thus, we reset the memory sequence to be a singleton sequence containing the last memory state prior to the crash.

Following the design of TXPMDK (and our concurrent extensions PMDK-TML and PMDK-NOREC) we do not check for reads and writes to unallocated memory within the library and instead delegate such checks to the client. An execution of TXPMDK (as well as PMDK-TML and PMDK-NOREC) that accesses unallocated memory is assumed to be faulty. In particular, a read or write of unallocated memory induces a *fault* (rule (F)). Once a fault is triggered, the program counter of each transaction is set to "fault" and recovery is impossible.

$$\begin{array}{l}\mathsf{validIdx}(n, \mathsf{s}, \mathsf{mems}) \triangleq \mathsf{s.beginIdx} \leq n < |\mathsf{mems}| \land \mathsf{s.rdSet} \subseteq \mathsf{mems}(n) \\ \qquad\qquad \land\ \mathsf{s.alSet} \subseteq \{l \mid \mathsf{mems}(n)(l) = \bot\}\end{array}$$

$$\frac{\begin{array}{c}\mathsf{PC}(t), \mathsf{S}(t), \mathsf{mems} \xrightarrow{\alpha} \mathsf{pc}, \mathsf{s}, \mathsf{mems}' \\ \mathsf{pc} \neq \text{fault}\end{array}}{\begin{array}{l}\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\alpha_t} \\ \mathsf{PC}[t \mapsto \mathsf{pc}], \mathsf{S}[t \mapsto \mathsf{s}], \mathsf{mems}'\end{array}}\ (\mathrm{S}) \qquad \frac{\begin{array}{c}\mathsf{PC}(t), \mathsf{S}(t), \mathsf{mems} \xrightarrow{\mathit{fault}} \text{fault}, \mathsf{s}, \mathsf{mems}' \\ \mathsf{PC}' = \lambda t.\text{fault}\end{array}}{\begin{array}{l}\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\mathit{fault}} \\ \mathsf{PC}', \mathsf{S}[t \mapsto \mathsf{s}], \mathsf{mems}'\end{array}}\ (\mathrm{F})$$

$$\frac{\begin{array}{c}\mathsf{PC}' = \lambda t.\ \mathtt{if}\ \mathsf{PC}(t) \notin \{\text{init}, \text{committed}, \text{fault}\} \\ \mathtt{then}\ \text{aborted}\ \mathtt{else}\ \mathsf{PC}(t)\end{array}}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\lightning} \mathsf{PC}', \mathsf{S}, \langle \mathit{last}(\mathsf{mems}) \rangle}\ (\mathrm{X}) \qquad \frac{\mathsf{PC} = \lambda t.\text{fault}}{\mathsf{PC}, \mathsf{S}, \mathsf{mems} \xrightarrow{\alpha_t} \mathsf{PC}, \mathsf{S}, \mathsf{mems}}\ (\mathrm{C})$$

$$(\mathrm{IB})\ \frac{\mathsf{pc} = \text{init}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{inv}(\mathtt{TxBegin})} \\ \Delta(\mathtt{TxBegin}), \mathsf{s}', \mathsf{mems}\end{array}} \qquad (\mathrm{DB})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxBegin}) \\ \mathsf{s}' = \mathsf{s}[\mathsf{beginIdx} \mapsto |\mathsf{mems}| - 1]\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxBegin})} \\ \text{ready}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{IOp})\ \frac{\mathsf{pc} = \text{ready} \quad a \in \mathit{InvOps}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{inv}(a)} \\ \Delta(a), \mathsf{s}, \mathsf{mems}\end{array}}$$

$$(\mathrm{DR\text{-}E})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxRead}(l)) \\ l \notin \mathsf{s.alSet} \cup \mathit{dom}(\mathsf{s.wrSet}) \\ \mathsf{validIdx}(n, \mathsf{s}, \mathsf{mems}) \\ \mathsf{mems}(n)(l) = v \qquad v \neq \bot \\ \mathit{rs} = \mathsf{s.rdSet} \oplus \{l \mapsto v\}\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxRead}(l,v))} \\ \text{ready}, \mathsf{s}[\mathsf{rdSet} \mapsto \mathit{rs}], \mathsf{mems}\end{array}} \qquad (\mathrm{FR})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxRead}(l)) \\ l \notin \mathsf{s.alSet} \cup \mathit{dom}(\mathsf{s.wrSet}) \\ \mathsf{validIdx}(n, \mathsf{s}, \mathsf{mems}) \\ \mathsf{mems}(n)(l) = \bot\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{fault}} \\ \text{fault}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{RA})\ \frac{\mathsf{pc} \notin \left\{\begin{array}{l}\text{init}, \text{ready}, \\ \text{committed}, \\ \text{aborted}, \text{fault}\end{array}\right\}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{Abort})} \\ \text{aborted}, \mathsf{s}, \mathsf{mems}\end{array}}$$

$$(\mathrm{DR\text{-}I})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxRead}(l)) \\ l \in \mathit{dom}(\mathsf{s.wrSet}) \\ \mathsf{s.wrSet}(l) = v\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxRead}(l,v))} \\ \text{ready}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{DR\text{-}A})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxRead}(l)) \\ l \notin \mathit{dom}(\mathsf{s.wrSet}) \\ l \in \mathsf{s.alSet}\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxRead}(l,0))} \\ \text{ready}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{DW})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxWrite}(l, v)) \\ l \in \mathsf{s.alSet} \lor \mathit{last}(\mathsf{mems})(l) \neq \bot \\ \mathit{ws} = \mathsf{s.wrSet} \oplus \{l \mapsto v\}\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxWrite}(l,v))} \\ \text{ready}, \mathsf{s}[\mathsf{wrSet} \mapsto \mathit{ws}], \mathsf{mems}\end{array}}$$

$$(\mathrm{FW})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxWrite}(l, v)) \\ l \notin \mathsf{s.alSet} \\ \mathit{last}(\mathsf{mems})(l) = \bot\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{fault}} \\ \text{fault}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{DA})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxAlloc}) \\ l \notin \mathsf{s.alSet} \\ \mathit{as} = \mathsf{s.alSet} \uplus \{l\}\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxAlloc}(l))} \\ \text{ready}, \mathsf{s}[\mathsf{alSet} \mapsto \mathit{as}], \mathsf{mems}\end{array}} \qquad (\mathrm{DC\text{-}RO})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxCommit}) \\ \mathsf{s.alSet} = \emptyset \\ \mathit{dom}(\mathsf{s.wrSet}) = \emptyset\end{array}}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\varepsilon} \\ \Pi(\mathtt{TxCommit}), \mathsf{s}, \mathsf{mems}\end{array}}$$

$$(\mathrm{RC})\ \frac{\mathsf{pc} = \Pi(\mathtt{TxCommit})}{\begin{array}{l}\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\mathit{res}(\mathtt{TxCommit})} \\ \text{committed}, \mathsf{s}, \mathsf{mems}\end{array}} \qquad (\mathrm{DC\text{-}W})\ \frac{\begin{array}{c}\mathsf{pc} = \Delta(\mathtt{TxCommit}) \\ \mathsf{validIdx}(\mathit{last}(\mathsf{mems}), \mathsf{s}, \mathsf{mems}) \\ \mathsf{mems}' = \mathsf{mems} +\!\!+ ((\mathit{last}(\mathsf{mems}) \oplus \{l \mapsto 0 \mid l \in \mathsf{s.alSet}\}) \oplus \mathsf{s.wrSet})\end{array}}{\mathsf{pc}, \mathsf{s}, \mathsf{mems} \xrightarrow{\varepsilon} \Pi(\mathtt{TxCommit}), \mathsf{s}, \mathsf{mems}'}$$

Fig. 8: The DDTMS global transitions (above) with its per-transaction transitions (below), where
$\mathit{InvOps} \triangleq \{\mathtt{TxWrite}(l, v), \mathtt{TxRead}(l) \mid l \in \textsc{Loc}, v \in \textsc{Val}\} \cup \{\mathtt{TxAlloc}, \mathtt{TxCommit}\}$

From a faulty state the system behaves chaotically, i.e. it is possible to generate any history using rule (C).

DDTMS Per-Transaction Transitions. The system contains externally visible transitions for *invoking* an operation (rules IB and IOP), which set the program counters to $\Delta(a)$, where a is the operation being performed. This allows the histories of the system to contain operation invocations without corresponding matching responses.

For the begin, allocation, read and write operations, an invocation can be followed by a single transition (rules DB, DA, DR-E, DR-I, DR-E and DW) that performs the operation combined with the corresponding *response*. Following an invocation, the commit operation is split into internal do actions ((DC-RO) and (DC-W)) and an external response (rule RC). Finally, after a read/write invocation, the system may perform a *fault transition* for a read (rule FR) or a write (rule FW). The main change from DTMS2 is the inclusion of an allocation procedure. The design of DDTMS allows the executing transaction, t, to tentatively allocate a location l within its transaction-local allocation set, $alSet_t$. This allocation in DDTMS is optimistic – correctness of the allocation is only checked when t performs a read or commits.

Successful (non-faulty) read and write operations take allocations into account as follows. **(1)** A read operation of transaction t reads from a prior write (rule (DR-I)) or allocation (rule (DR-A)) performed by t itself. In this case, the operation may only proceed if the location l is either in the allocation or write set of t. The effect of the operation is to return the value of l in the write set (if it exists) or 0 if it only exists in the allocation set. **(2)** A read operation of transaction t reads from a write or allocation performed by another transaction (rule (DR-E)). Note that as with DTMS2 and TMS2, in DDTMS a read-only transaction may serialise with any memory index n after $beginIdx_t$. Moreover, within validIdx, in addition to ensuring that t's read set is consistent with the memory index n (second conjunct), we must also ensure that t's allocation set is consistent with memory index n (third conjunct) by ensuring that none of the locations in the allocation set have been allocated at memory index n. **(3)** A write of transaction t successfuly performs its operation (rule (DW)), which can only happen if the location l being written has been allocated, either by t itself (first disjunct), or by a prior transaction (second disjunct). A writing transaction must serialise after the last memory index in mems, thus the second disjunct checks allocation against the last memory index.

A successful (non-faulty) transaction is split into two cases: **(1)** t is a read-only transaction (rule (DC-RO)), where both $alSet_t$ and $wrSet_t$ are empty for t. In this case, the transaction simply commits. **(2)** t has performed an allocation or a write (rule (DC-W)). Here, we check that t is valid with respect to the last memory in mems using validIdx. The commit introduces a new memory into the memory sequence mems. The update also ensures that all pending allocations in $alSet_t$ take effect before applying the writes from t's write set.

5.2 Soundness of ddTMS

We state our main theorem relating ddTMS to ddOpacity. As the models are inherently different, we need several definitions to transform ddTMS histories to those compatible with ddOpacity.

An *execution* of a labelled transition system (LTS) is an alternating sequence of states and actions, i.e. a sequence of the form $s_0\, a_1\, s_2\, a_2 \ldots s_{n-1}\, a_n\, s_n$ such that for each $0 < i \leq n$, $s_{i-1} \xrightarrow{a_i} s_i$ and s_0 is an initial state of the LTS. Suppose σ is an execution of ddTMS. We let $AH_\sigma = a_1\, a_2 \ldots a_n$ be the *action history* corresponding to σ, and EH_σ be the *external history* of σ, which is AH_σ restricted to non-ϵ actions. Let FF_σ be the longest *fault-free prefix* of EH_σ. We generate the history (in the sense of Def. 5) corresponding to FF_σ as follows. First, we construct the *labelled history*, LH_σ of σ from FF_σ by removing all invocation actions (leaving only responses and crashes). Then, we replace each response $a_i = \alpha_t$ by the event $(i, t, t, L(\alpha))$, where $L(res(\mathtt{TxBegin})) = \mathtt{B}$, $L(res(\mathtt{TxAlloc}(l))) = (\mathtt{M}, l, 0)$, $L(res(\mathtt{TxRead}(l, v))) = (\mathtt{R}, l, v)$, $L(res(\mathtt{TxWrite}(l, v))) = (\mathtt{W}, l, v)$, $L(res(\mathtt{Abort})) = \mathtt{A}$, $L(inv(\mathtt{TxCommit})) = \mathtt{C}$, and $L(res(\mathtt{TxCommit})) = \mathtt{S}$. Similarly, we replace each crash action $a_i = ↯$ by the pair $(i, ↯)$. Note that in this construction, for simplicity, we conflate threads and transactions, but this restriction is straightforward to generalise. Finally, let the *ordered history* of σ, denoted OH_σ, be the total order corresponding to LH_σ.

Theorem 2. *For any execution σ of ddTMS, the ordered history OH_σ satisfies ddOpacity.*

The definitions of (dynamic) durable opacity can lifted to the level of systems in the standard manner, providing a notion of correctness for implementations [28].

6 Modelling and Validating Correctness in FDR4

FDR4 [26] is a model checker for CSP [29] that has recently been used to verify linearisability [38], as well as opacity and durable opacity [23]. We similarly provide an FDR4 development, which allows proofs of refinement to be *automatically* checked up to certain bounds. This is in contrast to manual methods of proving correctness of concurrent objects [21,19], which require a significant amount of manual human input (though such manual proofs are unbounded).

An overview of our FDR4 development [47] is given in Fig. 9. We derive two specifications from ddTMS. The first is an FDR4 model of ddTMS itself, based on prior work [38,23], but contains the extensions described in §5.1. The second is ddTMS-Seq, which restricts ddTMS to a sequential crash-free specification. We use ddTMS-Seq to obtain (lower-bound) liveness-like guarantees, which strengthens traditional deadlock or divergence proofs of refinement. These lower-bound checks ensure our models contain at least the traces of ddTMS-Seq.

Fig. 10 summarises our experiments on the upper bound checks, where the times shown combine the compilation and model exploration times. Each row represents an experiment that bounds the number of transactions (#txns),

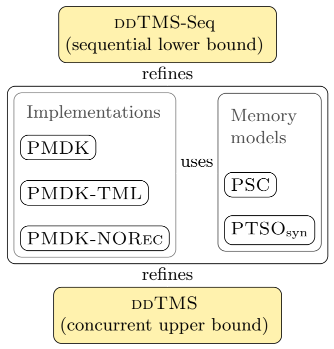


Fig. 9: Overview of FDR4 checks

Memory	#txns	#locs	#val	#buff	TX-PMDK	PMDK-TML	PMDK-NOREC
PSC	2	2	2	2	5.83s	5.90s	6.74s
PSC	2	3	2	2	201.03s	213.97s	271.35s
PSC	2	2	3	2	21.65s	23.47s	27.40s
PSC	2	2	2	3	5.83s	5.78s	6.60s
PTSO_{syn}	2	1	2	2	0.61s	3.96s	1.57s
PTSO_{syn}	2	2	2	2	6.67s	6.71s	7.73s
PTSO_{syn}	2	3	2	2	267.1s	268.91s	319.18s
PTSO_{syn}	2	2	3	2	24.10s	25.53s	29.24s
PTSO_{syn}	2	2	2	3	14.37s	14.19s	15.41s

Fig. 10: Summary of upper bounds checks (total time in seconds: compilation + model exploration). The time out (TO) is set to 1000 seconds of compilation time.

locations (#locs), values (#val) and the size of the persistency and store buffers (#buff). The times reported are for an Apple M1 device with 16GB of memory. The first row depicts a set of experiments where the implementations execute directly on NVM, without any buffers. As we discuss below, these tests are sufficient for checking lower bounds. The baseline for our checks sets the value of each parameter to two, and Fig. 10 allows us to see the cost of increasing each parameter. Note that all models time out when increasing the number of transactions to three, thus these times are not shown. Also note that for TXPMDK (which is single-threaded), the checks for PSC also cover PTSO_{syn}, since PTSO_{syn} is equivalent to PSC in the absence of races [31]. Nevertheless, it is interesting to run the single-threaded experiments on the PTSO_{syn} model to understand the impact of the memory model on the checks.

In our experiments we use FDR4's built-in *partial order reduction* features to make the upper bound checks feasible. This has a huge impact on the model checking speed; for instance, the check for PMDK-TML with two transactions, two locations, two values and buffer size of two reduces from over 6000 seconds (1 hour and 40 minutes) to under 7 seconds, which is almost a 1000-fold improvement! This speed-up makes it feasible to use FDR4 for rapid prototyping when developing programs that use TXPMDK, even for the relatively complex PTSO_{syn} memory model.

7 Related Work

Crash Consistency. Several authors have defined notions of atomicity for *concurrent objects* that take persistency into account (see [4] for a survey.) None of these conditions are suitable as they define consistency for concurrent operations (of concurrent data structures) as opposed to transactional memory.

Approaches and semantics to *crash-consistent* transactions stretch back to the mid 1970s, which considered the problem in the database setting [24,34]. Since then, a myriad of definitions have been developed for particular applications (e.g. distributed systems, file systems, etc.). For plain reads and writes, one of the first studies of persistency models focussed on NVM is by Pelley et al. [42]. Since then, several semantic models for real hardware (Intel and ARM) have been developed [50,49,31,12,48]. For transactional memory, there are only a few notions that combine a notion of crash consistency with ACID guarantees as required for concurrent durable transactions. Raad et al. [50] define a *persistent serializability* under relaxed memory, which does not handle aborted transactions. As we have already discussed, Bila et al. [6] define *durable opacity*, but this is defined in terms of (totally ordered) histories as opposed to partially ordered graphs. Neither persistent serialisability nor durable opacity handle allocation.

Validating the TXPMDK Implementation. Even without a clear consistency condition, a range of papers have explored correctness of the C/C++ implementation. Bozdogan et al. [8] built a sanitiser for persistent memory and used it to uncover memory-safety violations in TXPMDK. Fu et al. [25] have built a tool for testing persistent key-value stores and uncovered consistency bugs in the PMDK libraries. Liu et al. [36] have built a tool for detecting cross-failure races in persistent programs, and uncovered a bug in PMDK's libpmemobj library (see 'Bug 4' in their paper). They are at a different level of abstraction than ours since they focus on the code itself and do not provide any description of the design principles behind PMDK.

Raad et al. [45] and Bila et al. [7] have developed logics for reasoning about programs over the Px86-TSO model (which we recall is equivalent to $\mathrm{PTSO}_{\mathrm{syn}}$). However, these logics have thus far only been applied to small examples. Extending these logics to cover a proof by simulation and a full (manual) proof of correctness of PMDK, PMDK-TML and PMDK-NOREC would be a significant undertaking, but an interesting avenue for future work.

Transactional Memory (TM). Several works have studied the semantics of TM [15,22,44,43]. However, our works differ from those in that they do not account for persistency guarantees and crash consistency. However, while earlier works [44,43] merely *propose* a model for weak isolation (i.e. mixing transactional and non-transactional accesses), [15,22] formalise the weak isolation in various hardware and software TM platforms, albeit without validating their semantics.

Several approaches to crash consistency have recently been proposed. For a survey and comparison of techniques (in addition to transactions) see [3]. OneFile [52], Romulus [16], and Trinity and Quadra [51] together describe a set of algorithms that aim to improve the efficiency of TXPMDK by reducing the number of fence instructions. Liu et al. [35] present DudeTM, a persistent TM design that uses a shadow copy of NVM in DRAM, which is is shared amongst all transactions. Their approach comprises three key steps: Zardoshti et al. [56] present an alternative technique for making STMs persistent by instrumenting STM code with additional logging and flush instructions. However, none of these works have defined any formal correctness guarantees, and hence do not offer any proofs of correctness either. In particular, the role of allocation and its interaction with reads and writes is generally unclear.

As well as defining durable opacity, Bila et al. [6] develop a persistent version of the TML STM [17] by introducing explicit undo logging and flush instructions. They then prove this to be durably opaque via the dTMS2 specification. More recently, Bila et al. [5] have developed a technique for transforming both an STM and its corresponding opacity proof by delegating reads/writes to memory locations controlled by the TM to an abstract library that is later refined to use volatile and non-volatile memory. Neither of these works use txPMDK, and are over a sequentially consistent memory model.

8 Conclusions and Future Work

Our main contribution is validating the correctness for txPMDK via the development of declarative (ddOpacity) and operational (ddTMS) consistency criteria. We provide an abstraction of txPMDK and show that it satisfies ddTMS and hence ddOpacity by extension. Additionally, we develop PMDK-TML and PMDK-NORec as two concurrent extensions of txPMDK that are based on existing STM designs, and show that these also satisfy ddTMS (and hence ddOpacity). All of our models are validated under the PSC and $\mathrm{PTSO}_{\mathrm{syn}}$ memory models using FDR4.

As with most accepted existing transactional models (be it with or without persistency), we assume *strong isolation*, where each non-transactional access behaves like a singleton transaction (a transaction with a single access). That is, even ignoring persistency, there are no accepted definitions or models for mixing non-transactional and transactional accesses, and all existing transactional models (including opacity and serialisability) assume strong isolation. Indeed, PMDK transactions are specifically designed to be used in a purely transactional setting and are not meant to be used in combination with non-transactional accesses; i.e. they would have undefined semantics otherwise. Consequently, as we do not consider mixing transactional code with non-transactional code, RMW (read-modify-write) instructions are irrelevant in our setting. Specifically, as non-transactional access are treated as singleton transactions, RMW instructions are not needed or relevant since they behave as transactions and their atomicity would be guaranteed by the transactional semantics.

One threat to validity of our work is that the model checking results are on a small number of transactions, locations, values, and buffer sizes (see Fig. 10). However, we have found that these sizes have been adequate for validating all of our examples, i.e., when errors are deliberately introduced, FDR validation fails and counter-examples are automatically generated. Currently, we do not know whether there is a small model theorem for durable opacity in general. This is a separate line of work and a general question that we believe is out of the scope of this paper. Specifically, our focus here is on making PMDK transactions concurrent, providing a clear specification for PMDK (and its concurrent variations) with dynamic allocation, and validating correctness of the results under a realistic memory model.

References

1. Armstrong, A., Dongol, B., Doherty, S.: Proving opacity via linearizability: A sound and complete method. In: Bouajjani, A., Silva, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 50–66. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_4
2. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: Safety of live transactions in transactional memory: TMS is necessary and sufficient. In: Kuhn, F. (ed.) DISC. Lecture Notes in Computer Science, vol. 8784, pp. 376–390. Springer (2014). https://doi.org/10.1007/978-3-662-45174-8_26
3. Baldassin, A., Barreto, J., Castro, D., Romano, P.: Persistent memory: A survey of programming support and implementations. ACM Comput. Surv. **54**(7), 152:1–152:37 (2022). https://doi.org/10.1145/3465402
4. Ben-David, N., Friedman, M., Wei, Y.: Survey of persistent memory correctness conditions. CoRR **abs/2208.11114** (2022). https://doi.org/10.48550/ARXIV.2208.11114
5. Bila, E., Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Modularising verification of durable opacity. Log. Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/LMCS-18(3:7)2022
6. Bila, E., Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Defining and verifying durable opacity: Correctness for persistent software transactional memory. In: Gotsman, A., Sokolova, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 12136, pp. 39–58. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_3
7. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based Owicki-Gries reasoning for persistent x86-TSO. In: Sergey, I. (ed.) ESOP. Lecture Notes in Computer Science, vol. 13240, pp. 234–261. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_9
8. Bozdogan, K.K., Stavrakakis, D., Issa, S., Bhatotia, P.: SafePM: a sanitizer for persistent memory. In: Bromberg, Y., Kermarrec, A., Kozyrakis, C. (eds.) EuroSys. pp. 506–524. ACM (2022). https://doi.org/10.1145/3492321.3519574
9. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. pp. 55–64 (2016)
10. Cerone, A., Gotsman, A., Yang, H.: Transaction chopping for parallel snapshot isolation. In: DISC. vol. 9363, pp. 388–404 (2015)
11. Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M.F., Zeldovich, N.: GoJournal: a verified, concurrent, crash-safe journaling system. In: Brown, A.D., Lorch, J.R. (eds.) OSDI. pp. 423–439. USENIX Association (2021)
12. Cho, K., Lee, S., Raad, A., Kang, J.: Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In: Freund, S.N., Yahav, E. (eds.) PLDI. pp. 16–31. ACM (2021). https://doi.org/10.1145/3453483.3454027
13. Choe, J.: Review and things to know: Flash memory summit 2022. TechInsights (August 2022), https://www.techinsights.com/blog/review-and-things-know-flash-memory-summit-2022
14. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 211–225. ACM (2018). https://doi.org/10.1145/3192366.3192373

15. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, power, arm, and C++. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 211–225. ACM (2018). https://doi.org/10.1145/3192366.3192373
16. Correia, A., Felber, P., Ramalhete, P.: Romulus: Efficient algorithms for persistent transactional memory. In: Scheideler, C., Fineman, J.T. (eds.) Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018. pp. 271–282. ACM (2018). https://doi.org/10.1145/3210377.3210392
17. Dalessandro, L., Dice, D., Scott, M.L., Shavit, N., Spear, M.F.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M.R., Talia, D. (eds.) Euro-Par. Lecture Notes in Computer Science, vol. 6272, pp. 2–13. Springer (2010). https://doi.org/10.1007/978-3-642-15291-7_2
18. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining STM by abolishing ownership records. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPoPP. pp. 67–78. ACM (2010). https://doi.org/10.1145/1693453.1693464
19. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Mechanized proofs of opacity: a comparison of two techniques. Formal Aspects Comput. **30**(5), 597–625 (2018). https://doi.org/10.1007/s00165-017-0433-3
20. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects Comput. **25**(5), 769–799 (2013). https://doi.org/10.1007/s00165-012-0225-8
21. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015). https://doi.org/10.1145/2796550
22. Dongol, B., Jagadeesan, R., Riely, J.: Transactions in relaxed memory architectures. Proc. ACM Program. Lang. **2**(POPL) (Dec 2018). https://doi.org/10.1145/3158106
23. Dongol, B., Le-Papin, J.: Checking opacity and durable opacity with FDR. In: Calinescu, R., Pasareanu, C.S. (eds.) SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 222–242. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_13
24. Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976). https://doi.org/10.1145/360363.360369
25. Fu, X., Kim, W., Shreepathi, A.P., Ismail, M., Wadkar, S., Lee, D., Min, C.: Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In: van Renesse, R., Zeldovich, N. (eds.) SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021. pp. 100–115. ACM (2021). https://doi.org/10.1145/3477132.3483556
26. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
27. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010). https://doi.org/10.2200/S00253ED1V01Y201009DCT004
28. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
29. Hoare, C.A.R.: Communicating sequential processes (reprint). Commun. ACM **26**(1), 100–106 (1983). https://doi.org/10.1145/357980.358021
30. Intel: Persistent memory development kit, `libpmemobj` library (2022), https://pmem.io/pmdk/libpmemobj/

31. Khyzha, A., Lahav, O.: Taming x86-TSO persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434328
32. Krishnan, R.M., Kim, J., Mathew, A., Fu, X., Demeri, A., Min, C., Kannan, S.: Durable transactional memory can scale with timestone. In: ASPLOS. p. 335–349. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3373376.3378483
33. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
34. Lien, Y.E., Weinberger, P.J.: Consistency, concurrency and crash recovery. In: Lowenthal, E.I., Dale, N.B. (eds.) ACM SIGMOD International Conference on Management of Data. pp. 9–14. ACM (1978). https://doi.org/10.1145/509252.509258
35. Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., Ren, J.: Dudetm: Building durable transactions with decoupling for persistent memory. In: Chen, Y., Temam, O., Carter, J. (eds.) Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. pp. 329–343. ACM (2017). https://doi.org/10.1145/3037697.3037714
36. Liu, S., Seemakhupt, K., Wei, Y., Wenisch, T.F., Kolli, A., Khan, S.M.: Cross-failure bug detection in persistent memory programs. In: Larus, J.R., Ceze, L., Strauss, K. (eds.) ASPLOS. pp. 1187–1202. ACM (2020). https://doi.org/10.1145/3373376.3378452
37. Liu, S., Wei, Y., Zhao, J., Kolli, A., Khan, S.M.: PMTest: A fast and flexible testing framework for persistent memory programs. In: Bahar, I., Herlihy, M., Witchel, E., Lebeck, A.R. (eds.) ASPLOS. pp. 411–425. ACM (2019). https://doi.org/10.1145/3297858.3304015
38. Lowe, G.: Analysing lock-free linearizable datatypes using CSP. In: Gibson-Robinson, T., Hopcroft, P.J., Lazic, R. (eds.) Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10160, pp. 162–184. Springer (2017). https://doi.org/10.1007/978-3-319-51046-0_9
39. Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., Swanson, S.: Atomic in-place updates for non-volatile main memories with Kamino-Tx. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 499–512. EuroSys '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064215
40. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27
41. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (oct 1979). https://doi.org/10.1145/322154.322158
42. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. IEEE Micro **35**(3), 125–131 (2015). https://doi.org/10.1109/MM.2015.46
43. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 940–967. Springer International Publishing, Cham (2018)
44. Raad, A., Lahav, O., Vafeiadis, V.: On the semantics of snapshot isolation. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 1–23. Springer International Publishing, Cham (2019)

45. Raad, A., Lahav, O., Vafeiadis, V.: Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA), 151:1–151:28 (2020). https://doi.org/10.1145/3428219
46. Raad, A., Lahav, O., Wickerson, J., Balcer, P., Dongol, B.: Intel PMDK transactions: Specification, validation and concurrency (Extended version) (2023), https://doi.org/10.48550/arXiv.2312.13828
47. Raad, A., Lahav, O., Wickerson, J., Balcer, P., Dongol, B.: Intel PMDK transactions: Specification, validation and concurrency (Artifact) (2024). https://doi.org/10.6084/m9.figshare.24988173.v1
48. Raad, A., Maranget, L., Vafeiadis, V.: Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498683
49. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1–11:31 (2020). https://doi.org/10.1145/3371079
50. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1–135:27 (2019). https://doi.org/10.1145/3360561
51. Ramalhete, P., Correia, A., Felber, P.: Efficient algorithms for persistent transactional memory. In: Lee, J., Petrank, E. (eds.) PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021. pp. 1–15. ACM (2021). https://doi.org/10.1145/3437801.3441586
52. Ramalhete, P., Correia, A., Felber, P., Cohen, N.: Onefile: A wait-free persistent transactional memory. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019. pp. 151–163. IEEE (2019). https://doi.org/10.1109/DSN.2019.00028
53. Samsung Electronics: Samsung electronics unveils far-reaching, next-generation memory solutions at flash memory summit (2022), https://bit.ly/samsung_flash_memory_summit
54. Scargall, S.: Programming Persistent Memory: A Comprehensive Guide for Developers. APress (2020). https://doi.org/10.1007/978-1-4842-4932-1_8
55. Upadhyayula, U.: Introduction to persistent memory allocator and transactions (2020), https://www.intel.com/content/www/us/en/developer/videos/introduction-to-persistent-memory-allocator-and-transactions.html
56. Zardoshti, P., Zhou, T., Liu, Y., Spear, M.F.: Optimizing persistent memory transactions. In: 28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019. pp. 219–231. IEEE (2019). https://doi.org/10.1109/PACT.2019.00025

Artifact Report: Intel PMDK Transactions: Specification, Validation and Concurrency*

Azalea Raad[1], Ori Lahav[2], John Wickerson[1], Piotr Balcer[3], and Brijesh Dongol[4](✉)

[1] Imperial College London, London, UK
[2] Tel Aviv University, Tel Aviv, Israel
[3] Intel, Gdansk, Poland
[4] University of Surrey, Guildford, UK
b.dongol@surrey.ac.uk

Abstract. This report extends §6 of the main paper by providing further details of the mechanisation effort.

1 Modelling and Validating Correctness in FDR4

FDR4 [4] is a model checker for CSP [5] that has recently been used to verify linearisability [7], as well as opacity and durable opacity [3]. We similarly provide an FDR4 development, which allows proofs of refinement to be *automatically* checked up to certain bounds. This is in contrast to manual methods of proving correctness of concurrent objects [2,1], which require a significant amount of manual human input (though such manual proofs are unbounded). FDR4 uses a variety of underlying model checking paradigms and partial-order reduction techniques [4], depending on the structure of the files to be verified. FDR4 builds on FDR3, but the exact implementation details of FDR4 are not publicly available since it is a commercial product (available for free academic use).

The CSP files corresponding to this report may be downloaded from [8].

1.1 Modelling Details

One of the most challenging aspects of the FDR4 development is the modelling work itself. Our algorithms execute over a shared memory, but the CSP formalism is based on *communicating processes* with no notion of shared states. Thus, for each location we must explicitly define *handler* processes that communicate through *channels* to update and return the values of components (e.g. the addresses, read/write sets) of each model. Moreover, the implementations (TXPMDK,

* Raad is funded by a UKRI fellowship MR/V024299/1, EPSRC grant EP/X037029/1 and VeTSS. Lahav is supported by the Israel Science Foundation (grant 814/22) and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811). Wickerson is funded by EPSRC grant EP/R006865/1. Dongol is funded by EPSRC grants EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, EP/R025134/2 and VeTSS.

PMDK-NORec and PMDK-TML), the specification (ddTMS) and underlying memory models (PSC and PTSO$_{\mathrm{syn}}$) we consider are non-trivial, significantly increasing the challenge of the modelling effort. Although constructing the models is challenging, once the models have been developed, they can be combined in a modular fashion. We have taken advantage of this feature to combine our implementations with different memory models during development. The combination of PMDK-TML and TML/NOrec also takes advantage of this modularity.

This modularity also means that our models are reusable. One could use our models to check other developments, e.g. those that use txPMDK to implement other failure-atomic data structures, or verify redesigns of txPMDK over different memory models. Specifically, we use a top-level CSP process (which may comprise an interleaved composition of processes for each transaction) to model the most general client. Each transaction process begins a transaction, and then calls an unbounded number of reads, writes and allocations at non-deterministically chosen locations and with non-deterministically chosen values. An in-flight transaction process may also non-deterministically choose to terminate by calling commit instead of calling a read, write or allocation. Each operation call produces an externally visible invocation event, and when the operation terminates, an externally visible response is generated. Some operations may respond with an abort, in which case the transaction process itself terminates.

Additionally, there is an externally visible crash event that synchronises with all processes. At the level of the abstraction (i.e. ddTMS), this simply terminates all in-flight transactions, and resets the memory sequence (as detailed by the rule (X)). At the level of the implementation, all in-flight transactions are terminated and additionally, the store and persistency buffers are cleared. This means that when execution resumes, the value of each location is taken from NVM. Immediately after a crash (and before any other processes are started), the recovery process corresponding to the algorithm is executed. Note that transaction identifiers are never reused.

We eschew further details of our FDR4 models since they are provided as supplementary material [8] and also refer the interested reader to other prior works [7,3].

1.2 Overview of Development

An overview of our FDR4 development is given in Fig. 1. We derive two specifications from ddTMS. The first is an FDR4 model of ddTMS itself, based on prior work [7,3], but contains the extensions required for ddTMS. The second is ddTMS-Seq, which restricts ddTMS to a sequential crash-free specification. We use ddTMS-Seq to obtain (lower-bound) liveness-like guarantees, which strengthens traditional deadlock or divergence proofs of refinement. These lower-bound checks ensure our models contain at least the traces of ddTMS-Seq.

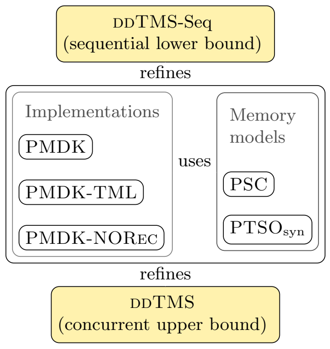


Fig. 1: Overview of FDR4 checks

Memory	#txns	#locs	#val	#buff	TXPMDK	PMDK-TML	PMDK-NOREC
PSC	2	2	2	2	5.83s	5.90s	6.74s
PSC	2	3	2	2	201.03s	213.97s	271.35s
PSC	2	2	3	2	21.65s	23.47s	27.40s
PSC	2	2	2	3	5.83s	5.78s	6.60s
PTSO$_{syn}$	2	1	2	2	0.61s	3.96s	1.57s
PTSO$_{syn}$	2	2	2	2	6.67s	6.71s	7.73s
PTSO$_{syn}$	2	3	2	2	267.1s	268.91s	319.18s
PTSO$_{syn}$	2	2	3	2	24.10s	25.53s	29.24s
PTSO$_{syn}$	2	2	2	3	14.37s	14.19s	15.41s

Fig. 2: Summary of upper bounds checks (total time in seconds: compilation + model exploration). The time out (TO) is set to 1000 seconds of compilation time.

CSP files. Our development comprises the following files.

File	Description
`Types.csp`	Contains the basic types and parameters. Use this file to increase / decrease the number of transactions, memory locations, values, etc. Defaults to 2 transactions, 2 locations and two values.
`MemoryP.csp`	Handler for memory, as well as the redo and undo logs. Operations query handlers to read/update the shared memory, flush to persistent memory and recover. This file is used to switch between memory models (NVM (which contains no crashes), PSC and PTSO$_{syn}$) - see the bottom of the file.
`LocHandler.csp`	Handler for local memory (i.e., the `loc` variable used by the implementations in Figs. 5 and 6.
`ddTMS.csp`	Model of the DDTMS automata from the main paper (Fig. 8).
`PMDK.csp`	Model of PMDK from Fig. 4 of the main paper.
`PMDK-TML.csp`	Model of PMDK-TML from Fig. 5 of the main paper.
`PMDK-NOrec.csp`	Model of PMDK-NOREC from Fig. 6 of the main paper.
`Refinement.csp`	File containing all checks to be performed.

Description of Tests. The file `Refinement.csp` comprises six tests as detailed in Figs. 9 and 10 of the paper. There are three upper-bound checks, which show that PMDK, PMDK-TML and PMDK-NOREC are refinements of DDTMS, validating soundness:

– `FinalTMS [T= PMDK`, checking that PMDK refines DDTMS.
– `FinalTMS [T= FinalTML`, checking that PMDK-TML refines DDTMS.
– `FinalTMS [T= FinalNOrec`, checking that PMDK-NOREC refines DDTMS.

Each of these tests can be run against the memory models: NVM (which contains no crashes), PSC and PTSO$_{syn}$ by commenting/uncommenting the relevant lines at the end of the file `MemoryP.csp`.

Additionally, there are three lower-bound checks, which show DDTMS-SEQ are refinements of PMDK, PMDK-TML and PMDK-NOREC.

- `PMDK [T= SeqFinalTMS`
- `FinalTML [T= SeqFinalTMS`
- `FinalNOrec [T= FinalNOrec`

Each of these tests can be run against the memory models: NVM and PSC as defined in the file `MemoryP.csp`. Note that the test against PTSO_{syn} times out. However, the tests above are sufficient since PTSO_{syn} reduces to PSC in the absence of data races (e.g., sequential executions).

Each check in FDR4 is split into two phases: **(1)** a compilation phase that builds the models; and **(2)** a model exploration phase. The characteristics of the upper and lower bounds checks are distinct. When naively checking the upper bound, compilation is almost instantaneous but model exploration times can be significant; these times are swapped for the lower bounds checks.

In general, lower-bounds take much longer to verify than the upper-bounds since FDR4 is optimised to verify abstract (low-detail) specifications are refined by concrete (high-detail) implementations. The lower bounds checks use the more complex models as the specification, leading to the creation of very large space-inefficient models, putting pressure on the available system memory. However, the lower-bound checks for PSC and PTSO_{syn} are superceded by the corresponding checks over NVM, since the memory models PSC and PTSO_{syn} are both supersets of NVM. That is, any trace over NVM must also be a trace PSC and PTSO_{syn}. For two transactions, two locations and two values, the checks for PMDK, PMDK-TML and PMDK-NORec take 12.16, 17.36, and 56.02 seconds, respectively.

1.3 Summary of Results

Fig. 2 summarises our experiments on the upper bound checks, where the times shown combine the compilation and model exploration times. Each row represents an experiment that bounds the number of transactions (#txns), locations (#locs), values (#val) and the size of the persistency and store buffers (#buff). The times reported are for an Apple M1 device with 16GB of memory. The first row depicts a set of experiments where the implementations execute directly on NVM, without any buffers. As we discuss below, these tests are sufficient for checking lower bounds. The baseline for our checks sets the value of each parameter to two, and Fig. 2 allows us to see the cost of increasing each parameter. Note that all models time out when increasing the number of transactions to three, thus these times are not shown. Also note that for txPMDK (which is single-threaded), the checks for PSC also cover PTSO_{syn}, since PTSO_{syn} is equivalent to PSC in the absence of races [6]. Nevertheless, it is interesting to run the single-threaded experiments on the PTSO_{syn} model to understand the impact of the memory model on the checks.

In our experiments we use FDR4's built-in *partial order reduction* features to make the upper bound checks feasible. This has a huge impact on the model checking speed; for instance, the check for PMDK-TML with two transactions, two locations, two values and buffer size of two reduces from over 6000 seconds

(1 hour and 40 minutes) to under 7 seconds, which is almost a 1000-fold improvement! This speed-up makes it feasible to use FDR4 for rapid prototyping when developing programs that use TXPMDK, even for the relatively complex PTSO_{syn} memory model.

References

1. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Mechanized proofs of opacity: a comparison of two techniques. Formal Aspects Comput. **30**(5), 597–625 (2018). https://doi.org/10.1007/s00165-017-0433-3
2. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. **48**(2), 19:1–19:43 (2015). https://doi.org/10.1145/2796550
3. Dongol, B., Le-Papin, J.: Checking opacity and durable opacity with FDR. In: Calinescu, R., Pasareanu, C.S. (eds.) SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 222–242. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_13
4. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 187–201 (2014)
5. Hoare, C.A.R.: Communicating sequential processes (reprint). Commun. ACM **26**(1), 100–106 (1983). https://doi.org/10.1145/357980.358021
6. Khyzha, A., Lahav, O.: Taming x86-tso persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434328
7. Lowe, G.: Analysing lock-free linearizable datatypes using CSP. In: Gibson-Robinson, T., Hopcroft, P.J., Lazic, R. (eds.) Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10160, pp. 162–184. Springer (2017). https://doi.org/10.1007/978-3-319-51046-0_9
8. Raad, A., Lahav, O., Wickerson, J., Balcer, P., Dongol, B.: Intel PMDK transactions: Specification, validation and concurrency (Artifact) (2024). https://doi.org/10.6084/m9.figshare.24988173.v1

Specifying and Verifying Persistent Libraries

Léo Stefanesco[1](✉), Azalea Raad[2], and Viktor Vafeiadis[1]

[1] MPI-SWS, Kaiserslautern, Germany
leo.stefanesco@mpi-sws.org
[2] Imperial College, London, United Kingdom

Abstract. We present a general framework for specifying and verifying *persistent libraries*, that is, libraries of data structures that provide some persistency guarantees upon a failure of the machine they are executing on. Our framework enables modular reasoning about the correctness of individual libraries (horizontal and vertical compositionality) and is general enough to encompass all existing persistent library specifications ranging from hardware architectural specifications to correctness conditions such as durable linearizability. As case studies, we specify the FliT and Mirror libraries, verify their implementations over Px86, and use them to build higher-level durably linearizable libraries, all within our framework. We also specify and verify a persistent transaction library that highlights some of the technical challenges which are specific to persistent memory compared to weak memory and how they are handled by our framework.

1 Introduction

Persistent memory (PM), also known as non-volatile memory (NVM), is a new kind of memory, which can be used to extend the capacity of regular RAM, with the added benefit that its contents are preserved after a crash (e.g. a power failure). Employing PM can boost the performance of any program with access to data that needs to survive power failures, be it a complex database or a plain text editor.

Nevertheless, doing so is far from trivial. Data stored in PM is mediated through the processors' caching hierarchy, which generally does not propagate all memory accesses to the PM in the order issued by the processor, but rather performs these accesses on the cache and only propagates them to the memory asynchronously when necessary (i.e. upon a cache miss or when the cache has reached its capacity limit). Caches, moreover, do not preserve their contents upon a power failure, which results in rather complex persistency models describing when and how stores issued by a program are guaranteed to survive a power failure. To ensure correctness of their implementations, programmers have to use low-level primitives, such as *flushes* of individual cache lines, *fences* that enforce ordering of instructions, and *non-temporal stores* that bypass the cache hierarchy.

These primitives are often used to implement higher-level abstractions, packaged into *persistent libraries*, i.e. collections of data structures that must guarantee to preserve their contents after a power failure. Persistent libraries can be

https://doi.org/10.1007/978-3-031-57267-8_8

thought of as the analogue of concurrent libraries for persistency. And just as concurrent libraries require a specification, so do persistent libraries.

The question naturally arises: what is the right specification for persistent libraries? Prior work has suggested a number of candidate definitions, such as *durable linearizability*, *buffered durable linearizability* [17], and *strict linearizability* [1], which are all extensions of the well-known correctness condition for concurrent data structures (i.e. linearizability [15]). In general, these definitions stipulate the existence of a total order among all executed library operations, a contiguous prefix of which is persisted upon a crash: the various definitions differ in exactly what this prefix should be, e.g. whether it is further constrained to include all fully executed operations.

Even though these specifications have a nice compositionality property, we argue that none of them are *the* right specification pattern for *every* persistent concurrent library. While for high-level persistent data structures, such as stacks and queues, a strong specification such as durable or strict linearizability would be most appropriate, this is certainly not the case for a collection of low-level primitives. Take, for instance, a library whose interface simply exposes the exact primitives of the underlying platform: memory accesses, fences and flushes. Their semantics, recently formalized in [30,19,28] in the case of the Intel-x86 architecture and in [31,5] in the case of the ARMv8 architecture, quite clearly do not fit into the framework of the durable linearizability definitions. More generally, there are useful concurrent libraries (especially in the context of weak memory consistency) that are not linearizable [26]; it is, therefore, conceivable that making those libraries persistent will require weak specifications.

Another key problem with attempting to specify persistent libraries *modularly* is that they often break the usual abstraction boundaries. Indeed, some models such as epoch persistency [6,24] provide a global persistency barrier that affects all memory locations, and therefore all libraries using them. Such global operations also occur at higher abstraction layers: persistent transactional libraries often require memory locations to be registered with the library in order for them to be used inside transactions. As such, to ensure compatibility with such transactional libraries, implementers of other libraries must register all locations they use and ensure that any component libraries they use do the same.

In this paper, we introduce a *general declarative framework* that addresses both of these challenges. Our framework provides a very flexible way of specifying persistent libraries, allowing each library to have a very different specification—be it durable linearizability or a more complex specification in the style of the hardware architecture persistency models. Further, to handle libraries that have a global effect (such as persistent barriers above) or, more generally, that make some assumptions about the internals of all other libraries, we introduce a *tag* system, allowing us to describe these assumptions *modularly*.

Our framework supports both *horizontal* and *vertical compositionality*. That is, we can verify an execution containing multiple libraries by verifying each library separately (horizontal compositionality). Moreover, we can completely verify the implementation of a library over a set of other libraries using the specifications of its constituent libraries without referring to their implementations (vertical compositionality). To achieve the latter, we define a semantic notion of substitution in terms of execution graphs, which replaces each library node by a suitably constrained set of nodes (its implementation).

For simplicity, in §2, we develop a first version of our framework over the classical notion of an execution *history* [15], which we extend with a notion of crashes. This basic version of our framework includes full support for weak persistency models but assumes an interleaving semantics of concurrency; i.e. sequential consistency (SC) [23].

Subsequently, in §3 we generalise and extend our framework to handle weak consistency models such as x86-TSO [32] and RC11 [22], thereby allowing us to represent hardware persistency models such as Px86 [30] and PARMv8 [31], in our framework. To do so, we rebase our formal development over execution graphs using Yacovet [26] as a means of specifying the consistency properties of concurrent libraries.

We illustrate the utility of our framework by encoding in it a number of existing persistency models, ranging from actual hardware models such as Px86 [30], to general-purpose correctness conditions such as durable linearizability [17]. We further consider two case studies, chosen to demonstrate the expressiveness of our framework beyond the kind of case studies that have been worked out in the consistency setting.

First, in §4 we use our framework to develop the first formal specifications of the FliT [35] and Mirror [10] libraries and establish the correctness of not only their implementations against their respective specifications, but also their associated constructions for turning a linearizable library into a durably linearizable one. This generic theorem is new compared to the case studies in [26], and leverages our 'semantic' approach in §3. Moreover, our proofs of these constructions are the *first* to establish this result in a weak consistency setting.

Second, in §5 we specify and prove an implementation of a persistent transactional library $\mathsf{L}_{\mathsf{trans}}$, which provides a high-level construction to persist a set of writes *atomically*. The $\mathsf{L}_{\mathsf{trans}}$ library illustrates the need for a 'well-formedness' specification (in addition to its consistency and persistency specifications) that requires clients of the $\mathsf{L}_{\mathsf{trans}}$ library to ensure e.g. that $\mathsf{L}_{\mathsf{trans}}$ writes appear only inside transactions. Moreover, it demonstrates the use of our tagging system to enable other libraries to interoperate with it.

Contributions and Outline. The remainder of this article is organised as follows.

- **§2** We present our general framework for specifying and verifying persistent libraries in the strong sequential consistency setting.
- **§3** We further generalise our framework to account for weaker consistency models.
- **§4** We use our framework to develop the first formal specifications of the FliT and Mirror libraries, verify their implementations against their specifications and prove their general construction theorems for turning linearizable libraries to durably linearizable ones.

§5 We specify a persistent transactional library $\mathsf{L}_{\mathsf{trans}}$, develop an implementation of $\mathsf{L}_{\mathsf{trans}}$ (over the Intel-x86 architecture) and verify it against its specification. We then consider two case studies of vertical and horizontal composition in our framework using $\mathsf{L}_{\mathsf{trans}}$.

We conclude and discuss related and future work in §6. The full proofs of all theorems stated in the paper are given in the technical appendix.

2 A General Framework for Persistency

We present our framework for specifying and verifying persistent *libraries*, which are collections of methods that operate on durable data structures. Following Herlihy et al. [15], we will represent program histories over a collection of libraries Λ as Λ-histories, i.e. as sequences of calls to the methods of Λ, which we will then gradually enhance to model persistency semantics. Throughout this section, we assume an underlying sequential consistency semantics; in §3 we will generalize our framework to account for weaker consistency models.

In the following, we assume the following infinite domains: **Meth** of method names, **Loc** of memory locations, **Tid** of thread identifiers, and $\mathbf{Val} \supseteq \mathbf{Loc} \cup \mathbf{Tid}$ of values. We let m range over method names, x over memory locations, t over thread identifiers, and v over values. An optional value $v_\bot \in \mathbf{Val}_\bot$ is either a value $v \in \mathbf{Val}$ or $\bot \notin \mathbf{Val}$.

2.1 Library Interfaces

A *library interface* declares a set of method invocations of the form $m(\boldsymbol{v})$. Some methods are are designated as constructors; a constructor returns a location pointing to the new library instance (object), which is passed as an argument to other library methods. An interface additionally contains a function, loc, which extracts these locations from the arguments and return values of its method calls.

Definition 1. *A* library interface *L is a tuple $\langle \mathcal{M}, \mathcal{M}_{\mathrm{c}}, \mathrm{loc}\rangle$, where the set of method invocations $\mathcal{M}$ is a subset of $\mathcal{P}(\mathbf{Meth} \times \mathbf{Val}^*)$, $\mathcal{M}_{\mathrm{c}} \subseteq \mathcal{M}$ is the set of constructors, and* $\mathrm{loc} : \mathcal{M} \times \mathbf{Val}_\bot \to \mathcal{P}(\mathbf{Loc})$ *is the location function.*

Example 1 (Queue library interface). The queue library interface, $\mathsf{L}_{\mathsf{Queue}}$, has three methods: a constructor $\mathsf{QueueNew}()$, which returns a new empty queue; $\mathsf{QueueEnq}(x, v)$ which adds value v to the end of queue x; and $\mathsf{QueueDeq}(x)$ which removes the head entry in queue x. We define $\mathrm{loc}(\mathsf{QueueNew}(), x) = \mathrm{loc}(\mathsf{QueueEnq}(x, \_), \_) = \mathrm{loc}(\mathsf{QueueDeq}(x), \_) = \{x\}$.

A *collection* Λ is a set of library interfaces with disjoint method names. When Λ consists of a single library interface L, we often write L instead of $\{L\}$.

2.2 Histories

Given a collection Λ, an event $e \in \textbf{Events}(\Lambda)$ of Λ is either a method invocation $m(\boldsymbol{v})_{\mathsf{t}}$ with $m(\boldsymbol{v}) \in \bigcup_{L \in \Lambda} L.\mathcal{M}$ and $\mathsf{t} \in \textbf{Tid}$ or method response (return) event $\textbf{ret}(v)_{\mathsf{t}}$.

A Λ-history is a sequence of events of Λ whose projection to each thread is an alternating sequence of invocation and return events which starts with an invocation.

Definition 2 (Sequential event sequences). *A sequence of events $e_1 \dots e_n$ is* sequential *if all its odd-numbered events $e_1, e_3, \dots$ are invocation events and all its even-numbered events $e_2, e_4, \dots$ are return events.*

Definition 3 (Histories). *A Λ-*history *is a finite sequence of events $H \in \textbf{Events}(\Lambda)^*$, such that for every thread t, the sub-sequence $H[\mathsf{t}]$ comprising only of t events is sequential. The set $\textbf{Hist}(\Lambda)$ denotes the set of all Λ-histories.*

When clear from the context, we refer to *occurrences* of events in a history by their corresponding events. For example, if $H = e_1 \dots e_n$ and $i < j$, we say that e_i *precedes* e_j and that e_j *succeeds* e_i. Given an invocation $m(\boldsymbol{v})_{\mathsf{t}}$ in H, its *matching return* (when it exists) is the first event of the form $\textbf{ret}(v)_{\mathsf{t}}$ that succeeds it (they share the same thread). A *call* is a pair $m(\boldsymbol{v})_{\mathsf{t}}{:}v_{\perp}$ of an invocation and either its matching return $v_{\perp} \in \textbf{Val}$ (*complete call*) or $v_{\perp} = \perp$ (*incomplete call*).

A *library* (specification) comprises an interface and a set of *consistent* histories. The latter captures the allowed behaviors of the library, which is a guarantee made by the library implementation.

Definition 4. *A* library specification *(or simply a* library*) L is a tuple $\langle L, \mathcal{S}_{\mathsf{c}} \rangle$, where L is a library interface, and $\mathcal{S}_{\mathsf{c}} \subseteq \textbf{Hist}(L)$ denotes its set of* consistent *histories.*

2.3 Linearizability

Linearizability [15] is a standard way of specifying concurrent libraries that have a sequential specification S, denoting a set of finite sequences of complete calls. Given a sequential specification S, a concurrent library L is linearizable under S if each consistent history of L can be *linearized* into a sequential one in S, while respecting the *happens before* order, which captures causality between calls. It is sufficient to consider consistent executions because inconsistent executions are, by definition, guaranteed by the library to never happen. Happens-before is defined as follows.

Definition 5 (Happens-Before). *A method call C_1* happens before *another method call C_2 in a history H, written $C_1 \prec_H C_2$ if the response of C_1 precedes the invocation of C_2 in H. When the choice of H is clear from the context, we drop the H subscript from $\prec$.*

A history H is *linearizable* under a sequential specification S if there exists a linearization (in the order-theoretic sense) of $\prec_H$ that belongs to S. The subtlety is the treatment of incomplete calls, which may or may not have taken effect. We write $\mathsf{compl}(H)$ for the set of histories obtained from a history H by appending zero or more matching return events. We write $\mathsf{trunc}(H)$ for the history obtained from H by removing its incomplete calls. We can then define linearizability as follows [14].

Definition 6. *A sequential history H_ℓ is a* sequentialization *of a history H if there exists $H' \in \mathsf{trunc}(\mathsf{compl}(H))$ such that H_ℓ is a linearization of $\prec_{H'}$. A history H is* linearizable under *S if there exists a sequentialization of H that belongs to S. A library* L *is* linearizable *under S if all its consistent histories are linearizable under S.*

For instance, we can specify the notion of *linearizable queues* as those that linearizable under the following sequential queue specification, S_{Queue}.

Example 2 (Sequential queue specification). The behaviors of a sequential queue, S_{Queue}, is expressed as a set of sequential histories as follows. Given a history H of L_{Queue} and a location $x \in \mathbf{Loc}$, let $H[x]$ denote the sub-history containing calls c such that $\mathrm{loc}(c) = \{x\}$. We define S_{Queue} as the set of all sequential histories H of L_{Queue} such that for all $x \in \mathbf{Loc}$, $H[x]$ is of the form $\mathsf{QueueNew}()_{\mathsf{t}_0}{:}x \; e_1 \; \cdots \; e_n$, where each $\mathsf{QueueDeq}$ call in $e_1 \; \cdots \; e_n$ returns the value of the k-th $\mathsf{QueueEnq}$ call, if it exists and precedes the $\mathsf{QueueDeq}$, where k is the number of preceding $\mathsf{QueueDeq}$ calls returning non-null values; otherwise, it returns null.

2.4 Adding Failures

Our framework so far does not support reasoning about persistency as it lacks the ability to describe the persistent state of a library after a failure. Our first extension is thus to extend the set of events of a collection, $\mathbf{Events}(\Lambda)$, with another type of event, a crash event $\lightning$.

Crash events allow us to specify the durability guarantees of a library. For instance, a library that does not persist any of its data may specify that a history with crash events is consistent if all of its sub-histories between crashes are (independently) consistent. In other words, in such a library, the method calls before a crash have no effect on the consistency of the history after the crash. We modify the definition of happens-before accordingly by treating it both as an invocation and a return event. We also assume that, after a crash, the thread ids of the new threads are distinct from that of all the pre-crash threads. For libraries that do persist their data, a useful generic specification is *durable linearizability* [17], defined as follows.

Definition 7. *Given a history H, let* $\mathsf{ops}(H)$ *denote the sub-history obtained from H by removing all its crash markers. A history H is* durably linearizable *under S if there exists a sequentialization H_ℓ of* $\mathsf{ops}(H)$ *such that $H_\ell \in S$.*

Intuitively, this ensures that operations persist before they return, and they persist in the same order as they take effect before a crash.

Although durable linearizability can specify a large range of persistent data-structures, it can be too strong. For example, consider a (memory) register library $\mathsf{L_{wreg}}$ that only guarantees that writes to the *same* location are persisted in the order they are observed by concurrent reads. The $\mathsf{L_{wreg}}$ methods comprise $\mathsf{RegNew}()$ to allocate a new register, $\mathsf{RegWrite}(x, v)$ to write v to register x, and $\mathsf{RegRead}(x)$ to read from register x. The sequential specification S_{wreg} is simple: once a register is allocated, a read R on x returns the latest value written to x, or 0 if R happens before all writes. The associated durable linearizability specification requires that writes be persisted in the linearization order; however, this is often not the case on existing hardware, e.g. in $\mathsf{Px86}$ (the Intel-x86 persistency model) [30].

A more relaxed and realistic specification would consider two linearizations of the events: the standard *volatile* order $\prec$ and a *persistent* order nvo expressing the order in which events are persisted. The next sections will handle this more refined model, this paragraph only gives a quick tastes of the kind of models that are implemented by hardware. To capture the same-location guarantees, we stipulate a per-location ordering on writes that is respected by both linearizations. Specifically, we require an ordering mo of the write calls such that for all locations x: 1) restricting mo to x, written mo_x, totally orders writes to x; and 2) $\mathrm{mo}_x \subseteq \prec$ and $\mathrm{mo}_x \subseteq \mathrm{nvo}$. Given a history H, we can then combine these two linearizations by using $\prec$ after the last crash and nvo before.

Formally, a history H with $n-1$ crashes can be decomposed into n (crash-free) *eras*; i.e. $H = H_1 \cdot \lightning \cdots \lightning \cdot H_n$ where each H_i is crash-free. Let us write $\prec_i$ for $\prec \cap (H_i \times H_i)$ and so forth. We then consider k-sequentializations of the form $H_\ell^k = H_\ell^{(1)} \cdots H_\ell^{(k-1)} \cdot H_\ell^{(k)}$, where $H_\ell^{(k)}$ is a sequentialization of E_k w.r.t. $\prec_k$ and $H_\ell^{(i)}$ is a sequentialization of E_i w.r.t. nvo_i, for $i < k$. We can now specify our weak register library as follows, where H comprises n eras:

$$H \in \mathsf{L_{wreg}}.\mathcal{S}_{\mathsf{c}} \iff \forall k \leq n.\ \exists H_\ell^k\ k\text{-seq. of } H.\ H_\ell^k \in S_{\mathsf{wreg}}$$

Example 3. The following history is valid according to this specification but not according to the durably linearizable one:

$$W_{\mathsf{t}_1}(x,1) \cdot W_{\mathsf{t}_2}(y,1) \cdot R_{\mathsf{t}_3}(y) \cdot \mathbf{ret}_{\mathsf{t}_3}(1) \cdot R_{\mathsf{t}_3}(x) \cdot \mathbf{ret}_{\mathsf{t}_3}(0) \cdot \lightning \cdot R_{\mathsf{t}_4}(y) \cdot \mathbf{ret}_{\mathsf{t}_4}(0) \cdot R_{\mathsf{t}_4}(x) \cdot \mathbf{ret}_{\mathsf{t}_4}(1)$$

While the writes to x ($W_{\mathsf{t}_1}(x,1)$) and y ($W_{\mathsf{t}_2}(y,1)$) are executing, thread t_3 observes the new value (1) of y but the old value (0) of x; i.e. $\prec$ must order $W_{\mathsf{t}_2}(y,1)$ before $W_{\mathsf{t}_1}(x,1)$. By contrast, after the crash the new value (1) of x but the old value of y (0) is visible; i.e. nvo must order the two writes in the opposite order to $\prec$ ($W_{\mathsf{t}_1}(x,1)$ before $W_{\mathsf{t}_2}(y,1)$).

Persist Instructions. The persistent registers described above are too weak to be practical, as there is no way to control how writes to different locations are persisted. In realistic hardware models such as $\mathsf{Px86}$, this control is afforded to the programmer using per-location *persist* instructions (e.g. `CLFLUSH`), ensuring that all writes on a location x persist before a write-back on x. Here, we consider a coarser (stronger) variant, denoted by PFENCE, that ensures that *all* writes (on *all* locations) that happen before a PFENCE are persisted. Later in §3 we describe how to specify the behavior of per-location persist operations.

Formally, we specify PFENCE by extending the specification of $\mathsf{L}_{\mathsf{wreg}}$ as follows: given history H, write call c_w and PFENCE event c_f, if $c_w \prec_H c_f$, then $(c_w, c_f) \in$ nvo.

Example 4. Consider the history obtained from example 3 by adding a PFENCE:

$$W_{t_1}(x,1) \cdot W_{t_2}(y,1) \cdot R_{t_3}(y) \cdot \mathbf{ret}_{t_3}(1) \cdot R_{t_3}(x) \cdot \mathbf{ret}_{t_3}(0) \cdot \mathsf{PFENCE}_{t_4}() \cdot \mathbf{ret}_{t_4}() \cdot \lightning \cdot R_{t_4}(y) \cdot \mathbf{ret}_{t_4}(0) \cdot R_{t_4}(x) \cdot \mathbf{ret}_{t_4}(1)$$

This history is no longer consistent according to the extended specification of $\mathsf{L}_{\mathsf{wreg}}$: as PFENCE has completed (returned), all its $\prec$-previous writes must have persisted and thus must be visible after the crash (which is not the case for $W_{t_2}(y,1)$).

2.5 Adding Well-formedness Constraints

Our next extension is to allow library specifications to constrain the *usage* of the library methods by the client of the library. For example, a library for a mutual exclusion lock may require that the "release lock" method is only called by a thread that previously acquired the lock and has not released it in between. Another example is a transactional library, which may require that transactional read and write methods are only called within transactions, i.e. between a "transaction-begin" and a "transaction-end" method call.

We call such constraints library *well-formedness* constraints, and extend the library specifications with another component, $\mathcal{S}_{\mathsf{wf}} \subseteq \mathbf{Hist}(L)$, which records the set of well-formed histories of the library. Ensuring that a program produces only well-formed histories of a certain library is an obligation of the clients of that library, so that the library implementation can rely upon well-formedness being satisfied.

2.6 Tags and Global Specifications

The goal of our framework is not only to specify libraries in isolation, but also to express how a library can enforce persistency guarantees across other libraries. For example, consider a library $\mathsf{L}_{\mathsf{trans}}$ for persistent transactions, where all operations wrapped within a transaction persist together *atomically*; i.e. either all or none of the operations in a transaction persist.

The $\mathsf{L}_{\mathsf{trans}}$ methods are: $\mathsf{PTNewReg}$ to allocate a register that can be accessed (read/written) within a transaction; $\mathsf{PTBegin}$ and PTEnd to start and end a transaction, respectively; $\mathsf{PTRead(x)}$ and $\mathsf{PTWrite(x, v)}$ to read from and write to $\mathsf{L}_{\mathsf{trans}}$ register x, respectively; and $\mathsf{PTRecover}$ to restore the atomicity of transactions whose histories were interrupted by a crash.

Consider the snippet below, where the PEnq(q, 33) (enqueuing 33 into persistent queue q) and PSetAdd(s, 77) (adding 77 to persistent set s) are wrapped within an $\mathsf{L}_{\mathsf{trans}}$ transaction and thus should take effect atomically and at the latest after the end of the call to PTEnd.

```
PTBegin();
  PEnq(q, 33);
  PSetAdd(s, 77);
PTEnd();
```

Such guarantees are not offered by existing hardware primitives e.g. on Intel-x86 or ARMv8 [30,31] architectures. As such, to ensure atomicity, the persistent queue and set implementations cannot directly use hardware reads/writes; rather, they must use those provided by the transactional library whose implementation could use e.g. an undo-log to provide atomicity.

Our framework as described so far cannot express such cross-library persistency guarantees. The difficulty is that the transactional library relies on other libraries using certain primitives. This, however, is against the spirit of *compositional specification*, which precludes the transactional library from referring to other libraries (e.g. the queue or set libraries). Specifically, there are two challenges. First, both well-formedness requirements and consistency guarantees of $\mathsf{L}_{\mathsf{trans}}$ must apply to *any* method call that is designed to use (transitively) the primitives of $\mathsf{L}_{\mathsf{trans}}$. Second, we must formally express atomicity ("all operations persist atomically"), without $\mathsf{L}_{\mathsf{trans}}$ knowing what it means for a method of an arbitrary library to persist. In other words, $\mathsf{L}_{\mathsf{trans}}$ needs to introduce an abstract notion of 'having persisted' for an operation, and guarantee that all methods in a transaction 'persist' atomically.

To remedy this, we introduce the notion of *tags*. Specifically, to address the first challenge, the transactional library provides the tag $\textsc{t}$ to designate those operations that are 'transaction-aware' and as such must be used inside a transaction. To address the second challenge, the transaction library provides the tag $\textsc{p}^{\mathsf{tr}}$, denoting an operation that has abstractly persisted. The specification of $\mathsf{L}_{\mathsf{trans}}$ then guarantees that all operations tagged with $\textsc{t}$ inside a transaction persist atomically, in that either they are all tagged with $\textsc{p}^{\mathsf{tr}}$ of none of them are. Dually, using the well-formedness condition, $\mathsf{L}_{\mathsf{trans}}$ requires that all operations tagged with $\textsc{t}$ appear inside a transaction. Note that as the persistent queue and set libraries tag their operations with $\textsc{t}$, verifying their implementations incurs related proof obligations; we will revisit this later when we formalize the notion of library implementations.

Remark 1 (Why bespoke persistency?). The reader may question why 'having persisted' is not a primitive notion in our framework, as in an existing model of Px86 [19] where histories track the set P of persisted events. This is because associating a Boolean ('having persisted') flag with an operation may not be sufficient to describe whether it has persisted. To see this, consider a library $\mathsf{L}_{\mathsf{pair}}$ with operations $\mathsf{Write}(x, l, r)$ (writing (l, r) to pair x), $\mathsf{Readl}(x)$ and $\mathsf{Readr}(x)$ (reading the left and right components of x, respectively). Suppose $\mathsf{L}_{\mathsf{pair}}$ is implemented by storing the left component in an $\mathsf{L_{trans}}$ register and the right component in a $\mathsf{L_{wreg}}$ register. The specification of $\mathsf{L_{pair}}$ would need to track the persistence of each component separately, and hence a single set P of persisted events would not suffice.

Let us see how libraries can use these tags in *global well-formedness and consistency specifications*. The dilemma is, on the one hand, the specification of $\mathsf{L_{trans}}$ needs to refer to events from other libraries, but on the other hand, it should not depend on other libraries to preserve encapsulation. Our idea is to *anonymize* these external events such that the global specification depends only on their relevant tags. A library should only rely on the tags it introduces itself, as well as the tags of the libraries it uses.

We now revisit several of our definitions to account for *tags* and *global specifications*. A library interface now additionally holds the tags it introduces as well as those it uses. For instance, the $\mathsf{L_{trans}}$ library described above depends on no tag and introduces tags $\textsc{t}$ and $\textsc{p}^{\mathsf{tr}}$.

Definition 8 (Interfaces). *An* interface *is a tuple* $L = \langle \mathcal{M}, \mathcal{M}_{\mathrm{c}}, \mathrm{loc}, \textsc{Tags}_{\mathrm{new}}, \textsc{Tags}_{\mathrm{dep}} \rangle$*, where* $\mathcal{M}$*,* $\mathcal{M}_{\mathrm{c}}$*, and* loc *are as in Def. 1,* $\textsc{Tags}_{\mathrm{new}}$ *is the set of tags* L *introduces, and* $\textsc{Tags}_{\mathrm{dep}}$ *is the set of tags* L *uses. The set of tags usable by* L *is* $\textsc{Tags}(L) \triangleq L.\textsc{Tags}_{\mathrm{new}} \cup L.\textsc{Tags}_{\mathrm{dep}}$.

We next define the notion of tagged method invocations (where a method invocation is associated with a set of tags). Hereafter, our notions of events, history (and so forth) use tagged method invocations (rather than methods invocations).

Definition 9. *Given a library interface* L*, a* tagged method invocation *is of the form* $m(\boldsymbol{v})^{T}_{\mathsf{t}} : v_{\perp}$*, where the new component is a set of tags* $T \subseteq \textsc{Tags}(L)$*.*

A *global specification* of a library interface L is a set of histories with some "anonymized" events. These are formalized using a designated library interface, $\star_L$ (with a single method $\star$), which can be tagged with any tag from $\textsc{Tags}(L)$.

Definition 10. *Given an interface* L*, the interface* $\star_L$ *is* $\langle \{\star\}, \emptyset, \emptyset, \emptyset, \textsc{Tags}(L) \rangle$.

Now, given any history $H \in \mathbf{Hist}(\{L\} \cup \Lambda)$, let $\pi_L(H) \in \mathbf{Hist}(\{L, \star_L\})$ denote the *anonymization* of H such that each non-L event e in H labelled with a method $m(\boldsymbol{v})^{T}_{\mathsf{t}} : v_{\perp}$ of $L' \in \Lambda$ is replaced with $\star^{T}_{\mathsf{t}}$ of $\star_L$ if $T \neq \emptyset$ and is discarded otherwise. It is then straightforward to extend the notion of libraries with global specifications as follows.

Definition 11. *A* library specification L *is a tuple* $\langle L, \Lambda_{\mathrm{tags}}, \mathcal{S}_{\mathsf{c}}, \mathcal{S}_{\mathsf{wf}}, \mathcal{T}_{\mathsf{c}}, \mathcal{T}_{\mathsf{wf}} \rangle$*, where* L*,* $\mathcal{S}_{\mathsf{c}}$ *and* $\mathcal{S}_{\mathsf{wf}}$ *are as in Def. 4;* $\mathcal{T}_{\mathsf{c}}$ *and* $\mathcal{T}_{\mathsf{wf}} \subseteq \mathbf{Hist}(\{L, \star_L\})$ *are the* globally consistent *and* globally well-formed *histories, respectively; and* Λ_{tags} *denotes the* tag-dependencies*, i.e. a collection of libraries that provide all tags that* L *uses:* $L.\textsc{Tags}_{\mathrm{dep}} \subseteq \bigcup_{\mathsf{L}' \in \Lambda_{\mathrm{tags}}} \mathsf{L}'.\textsc{Tags}_{\mathrm{new}}$*. Both* $\mathcal{T}_{\mathsf{wf}}$ *and* $\mathcal{T}_{\mathsf{c}}$ *contain the empty history.*

In the context of a history, we write $\lfloor \textsc{t} \rfloor$ for the set of events or calls tagged with the tag $\textsc{t}$ (we consider a return event tagged the same way as its unique matching invocation).

For the $\mathsf{L_{trans}}$ library, the *globally* well-formed set $\mathsf{L_{trans}}.\mathcal{T}_{\mathsf{wf}}$ comprises histories H such that for each thread t, $E[\mathsf{t}]$ restricted to $\mathsf{PTBegin}$, PTEnd and events of the form T-tagged events is of the form described by the regular expression $(\mathsf{PTBegin}.\lfloor \text{T} \rfloor^*.\mathsf{PTEnd})^*$. In particular, transaction nesting is disallowed in our simple $\mathsf{L_{trans}}$ library.

To define global consistency, we need to know when two operations are part of the same transaction. Given a history H, we define the *same-transaction* relation, strans, relating pairs of $e, e' \in \lfloor \text{T} \rfloor \cup \mathsf{PTEnd} \cup \mathsf{PTBegin}$ executed by the same thread t such that there is no $\mathsf{PTBegin}$ or PTEnd executed by t between them. The set $\mathsf{L_{trans}}.\mathcal{T}_{\mathsf{c}}$ of globally consistent histories contains histories H such that $\forall (e, e') \in \text{strans}, e \in \lfloor \text{P}^{\mathsf{tr}} \rfloor \Leftrightarrow e' \in \lfloor \text{P}^{\mathsf{tr}} \rfloor$, and all completed PTEnd calls are in $\lfloor \text{P}^{\mathsf{tr}} \rfloor$. Since the PTEnd call is related to all events inside its transaction, this specification does express that (1) a transaction persist by the time the call to PTEnd finishes and (2) all events persist *atomically*.

Finally, we need to define the local consistency predicate $\mathsf{L_{trans}}.\mathcal{S}_{\mathsf{c}}$ describing the behavior of the registers provided by $\mathsf{L_{trans}}$. This is where the we define the concrete meaning of 'having persisted' for these registers. Let S be the sequential specification of a register. Let $H \in \mathbf{Hist}(\mathsf{L_{trans}})$ be a history decomposed into k eras as $H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \dots \cdot \lightning \cdot H_k$. Then $H \in \mathsf{L_{trans}}.\mathcal{S}_{\mathsf{c}}$ iff all events are tagged with T, and there exists a $\prec$-linearization H_ℓ of $\big((H_1 \cdot \lightning \cdot H_2 \cdot \lightning \cdot \dots \cdot \lightning \cdot H_{k-1}) \cap \lfloor \text{P}^{\mathsf{tr}} \rfloor \big) \cdot H_k$ such that $H_\ell \in S$, where $\lfloor \text{P}^{\mathsf{tr}} \rfloor$ is the set of events of H tagged with P^{tr}. In other words, a write operation is seen after a crash iff it has persisted. The requirement that such operations must appear within transactions and the guarantee that they persist at the same time in a transaction are covered by the global specifications.

2.7 Library Implementations

We have described how to *specify* persistent libraries in our framework, and next describe how to *implement* persistent libraries. This is formalized by the judgment $\Lambda \vdash I : \mathsf{L}$, stating that I *is a correct implementation of library* L *and only uses calls in the collection of libraries* Λ. As usual in such 'layered' frameworks [13,26], the base layer, which represents the primitives of the hardware, is specified as a library, keeping the framework uniform. This judgement can be composed vertically as follows, where $I[I_L]$ denotes replacing the calls to library L in I with their implementations given by I_L (which in turn calls libraries Λ'):

$$\frac{\Lambda, \mathsf{L} \vdash I : \mathsf{L}' \quad \Lambda' \vdash I_L : \mathsf{L}}{\Lambda, \Lambda' \vdash I[I_L] : \mathsf{L}'}$$

As we describe later, this judgment denotes *contextual refinement* and is impractical to prove directly. We define a stronger notion that is *compositional* and more practical to use.

Definition 12 (Implementation). *Given a collection Λ of libraries and a library* L*, an* implementation I of L over Λ *is a map,* $I \; : \; \mathsf{L}.\mathcal{M} \times \mathbf{Val}_\perp \; \longrightarrow$

```
globals log := Q.new()
method PTNewReg() := alloc(1)
method PTRead(l) := read(l)
method PTWrite(l, v) :=
  Q.append(log, (l, v));
  write(l, v)
method PTBegin() := FENCE();
method PTEnd() :=
  Append(log, COMMITTED);
  FENCE()

method PTRecover() :=
  let w = Q.new() in
  while (x := Q.pop(log))
    if (x = COMMITTED)
      w = Q.new();
    else
      Q.append(w, x);
  while ((l, v) = Q.pop(log)) {
    write(l, v); }
```

Fig. 1. Implementation of $\mathsf{L}_{\mathsf{trans}}$

$\mathcal{P}(\mathbf{Hist}(\Lambda))$*, such that it is downward-closed: 1) if* $H \in I(m(\boldsymbol{v})_{\mathsf{t}}, v_\perp)$ *and* H' *is a prefix of* H*, then* $H' \in I(m(\boldsymbol{v}), \perp)$*; and 2) each* $I(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$ *history only contain events by thread* t*.*

Intuitively, $I(m(\boldsymbol{v}), v_\perp)$ contains the histories corresponding to a call $m(\boldsymbol{v})$ with outcome $v_\perp$, where $v_\perp = \perp$ denotes that the call has not terminated yet and $v_\perp = v \in \mathbf{Val}$ denotes the return value. Downward-closure means that an implementation contains all partial histories. We use a concrete programming language to write these implementations; its syntax and semantics are standard and given in the appendix [34].

For example, the implementation of $\mathsf{L}_{\mathsf{trans}}$ over $\mathsf{L}_{\mathsf{wreg}}$ and $\mathsf{L}_{\mathsf{Queue}}$ is given in Fig. 1. The idea is to keep an undo-log as a persistent queue that tracks the values of the variables *before* the transaction begins. At the end of a transaction, and after all its writes have persisted, we write the sentinel value COMMITTED to the log to indicate that the transaction was completed successfully. After a crash, the recovery routine PTRecover returns the undo-log and undoes the operations of *incomplete* transactions by writing their previous values.

Histories and Implementations. An implementation I of L over Λ is correct if for all histories $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ that use library L as well as those in Λ', and all histories H' obtained by replacing calls to L methods with their implementation in I, if H' is consistent, then so is H (it satisfies the L specification).

We define the action $H \cdot I$ of an implementation I on an abstract history H in a 'relational' way: $H' \in H \cdot I$ when we can *match* each operation $m'(\boldsymbol{v})$ in H' with some operation $f(m'(\boldsymbol{v}))$ in H in such a way that the collection $f^{-1}(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$ of operations corresponding to some call $m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp$ in H agrees with $I(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$.

Definition 13. *Let* I *be an implementation of* L *over* Λ*; let* $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ *and* $H' \in \mathbf{Hist}(\Lambda \cup \Lambda')$ *be two histories. Given a map* $f : \{1, \ldots, |H'|\} \to \{1, \ldots, |H|\}$*,* H' (I, f)-matches H *if the following hold:*

1. f *is surjective;*

2. *for all invocations of H, if $m(\boldsymbol{v})_{\mathsf{t}} \notin \mathsf{L}.\mathcal{M}$, then $f(m(\boldsymbol{v})_{\mathsf{t}}) = m(\boldsymbol{v})_{\mathsf{t}}$;*
3. *for all threads t, if e_1 precedes e_2 in $H'[\mathsf{t}]$, then $f(e_1)$ precedes $f(e_2)$ in $H[\mathsf{t}]$;*
4. *for all calls $m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp$ of H, the set $f^{-1}(m(\boldsymbol{v})_{\mathsf{t}})$ corresponds to a substring H'_m of $H'[\mathsf{t}]$ and $H'_m \in I(m(\boldsymbol{v})_{\mathsf{t}}{:}v_\perp)$, where $v_\perp$ is the (optional) return value of $m(\boldsymbol{v})_{\mathsf{t}}$ in H.*

The action *of I on a history H is defined as follows:*

$$H \cdot I \triangleq \{H' \mid \exists f.\ H'\ (I, f)\text{-}matches\ H\}.$$

Condition 1 ensures that all events of the abstract history are matched with an implementation event; condition 2 ensures that the events that do not belong to the library being implemented (L) are left untouched, and condition 3 ensures that the thread-local order of events in the implementation agrees with the one in the specification. The last condition (4) states that the events corresponding to the implementation of a call $m(\boldsymbol{v})$ are consecutive in the history of the executing thread t, and correspond to the implementation I.

Well-formedness and Consistency. Recall that libraries specify both how they should be used (*well-formedness*), and what they guarantee if used correctly (*consistency*). Using these specifications (expressed as sets of histories) to define implementation correctness is more subtle than one might expect. Specifically, if we view a program using a library L as a downward-closed set of histories in $\mathbf{Hist}(\mathsf{L})$, we cannot assume all its histories are in the set $\mathsf{L}.\mathcal{S}_{\mathsf{wf}}$ of well-formed histories, as the semantics of the program will contain *unreachable* traces (see [26]). To formalize reachability at a semantic level, we define *hereditary consistency*, stating that each step in the history was consistent, and thus the current 'state' is reachable.

Definition 14 (Consistency). *History $H \in \mathbf{Hist}(\Lambda)$ is* consistent *if for all $\mathsf{L} \in \Lambda$, $H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_c$ and $\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_c$. It is* hereditarily consistent *if all $H[1..k]$ are consistent, for $k \leq |H|$.*

This definition uses the 'anonymization' operator π_{L} defined in §2.6 to test that the history H follows the global consistency predicates of every $\mathsf{L} \in \Lambda$.

We further require that programs using libraries respect *encapsulation*, defined below, stating that locations obtained from a library constructor are only used by that library instance. Specifically, the first condition ensures that distinct constructor calls return distinct locations. The second condition ensures that a non-constructor call e of L uses locations that have been allocated by an earlier call c ($c \prec e$) to an L constructor.

Definition 15 (Encapsulation). *A history $H \in \mathbf{Hist}(\Lambda)$ is* encapsulated *if the following hold, where C denotes the set of calls to constructors in H:*

1. *for all $c, c' \in C$, if $c \neq c'$, then $\mathrm{loc}(c) \cap \mathrm{loc}(c') = \emptyset$;*
2. *for all $e \in H \setminus C$, if $\mathrm{loc}(e) \neq \emptyset$, then there exist $c \in C$, $\mathsf{L} \in \Lambda$ such that $e, c \in \mathsf{L}.\mathcal{M}$, $c \prec e$ and $\mathrm{loc}(e) \subseteq \mathrm{loc}(c)$.*

We can now define when a history of Λ is *immediately well-formed*: it must be encapsulated and be well-formed according to each library in Λ and all the tags it uses.

Definition 16. *History $H \in \mathbf{Hist}(\Lambda)$ is* immediately well-formed *if the following hold:*

1. *H is encapsulated;*
2. *$H[\mathsf{L}] \in \mathsf{L}.\mathcal{S}_{\mathit{wf}}$, for all $\mathsf{L} \in \Lambda$; and*
3. *$\pi_{\mathsf{L}}(H) \in \mathsf{L}.\mathcal{T}_{\mathit{wf}}$ for all $\mathsf{L} \in \mathsf{TagDep}(\Lambda)$, where the immediate dependencies $\mathsf{TagDep}(\Lambda)$ is defined as $\bigcup_{\mathsf{L}\in\Lambda}\{\mathsf{L}\} \cup \Lambda_{\text{tags}}(\mathsf{L})$.*

We finally have the notions required to define a *correct implementation.*

Implementation Correctness. As usual, an implementation is correct if all behaviors of the implementation are allowed by the specification. In our setting, this means that if a concrete history is *hereditarily consistent*, so should the abstract history. Moreover, assuming the abstract history is well-formed, all corresponding concrete histories should also be well-formed; this corresponds to the requirement that the library implementation uses its dependencies correctly, under the assumption that the program itself uses its libraries correctly.

Definition 17 (Correct implementation). *An implementation I of L over Λ is* correct*, written $\Lambda \vdash I : \mathsf{L}$, if for all collections Λ', all 'abstract' histories $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda')$ and all 'concrete' histories $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda \cup \Lambda')$, the following hold:*

1. *if H is immediately well-formed, then H' is also immediately well-formed; and*
2. *if H' is immediately well-formed and hereditarily consistent, then H is consistent.*

This definition is similar to *contextual refinement* in that it quantifies over all contexts: it considers histories that use arbitrary libraries as well as those that concern I directly. We now present a more convenient, *compositional* method for proving an implementation correct, which allows one to only consider libraries and tags that are used by the implemented library.

2.8 Compositionally Proving Implementation Correctness

Recall that in this section we present our framework in a simplified sequentially consistent setting; later in §3 we generalize our framework to the weak memory setting. We introduce the notion of *compositional correctness*, simplifying the global correctness conditions in Def. 17. Specifically, while Def. 17 considers histories with arbitrary libraries that may use tags introduced by L, our compositional condition requires one to prove that only those L methods that are L-tagged satisfy $\mathsf{L}.\mathcal{T}_{\mathsf{c}}$.

Definition 18 (Compositional correctness). *An implementation I of* L *over Λ is* compositionally correct *if the following hold:*

1. *For all Λ', $H \in \mathbf{Hist}(\{\mathsf{L}\} \cup \Lambda)$ and $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda \cup \Lambda')$, if H' is well-formed, then H is well-formed;*
2. *For all $H \in \mathbf{Hist}(\mathsf{L})$ and $H' \in H \cdot I \subseteq \mathbf{Hist}(\Lambda)$, if H' is well-formed and hereditarily consistent, then $H \in \mathsf{L}.\mathcal{S}_{\mathsf{c}} \cap \mathsf{L}.\mathcal{T}_{\mathsf{c}}$; and*
3. *For all $\mathsf{L}' \in \Lambda$, $H \in \mathbf{Hist}(\{\mathsf{L}, \mathsf{L}', \star_{\mathsf{L}'}\})$ and $H' \in H \cdot I$, if $\pi_{\mathsf{L}'}(H') \in \mathsf{L}'.\mathcal{T}_{\mathsf{wf}} \cap \mathsf{L}'.\mathcal{T}_{\mathsf{c}}$, then $\pi_{\mathsf{L}'}(H) \in \mathsf{L}'.\mathcal{T}_{\mathsf{c}}$.*

The preservation of well-formedness (condition 1) does not change compared to its counterpart in Def. 17, as in practice this condition is easy to prove directly. Condition 2 requires one to prove that the implementation is correct *in isolation* (without Λ'). Condition 3 requires one to prove that global consistency requirements are maintained for all dependencies of the implementation. In practice, this corresponds to proving that those L operations tagged with existing tags in Λ obey the global specifications associated with these tags. Intuitively, the onus is on the library that *uses* a tag for its methods to prove the associated global consistency predicate: we need not consider unknown methods tagged with tags in $\mathsf{L}.\text{TAGS}_{\text{new}}$.

Finally, we show that it is sufficient to show an implementation I is compositionally correct as it implies that I is correct.

Theorem 1 (Correctness). *If an implementation I of* L *over Λ is* compositionally correct *(Def. 18), then it is also* correct *(Def. 17).*

Example 5 (Transactional Library $\mathsf{L}_{\mathsf{trans}}$). Consider the implementation I_{trans} of $\mathsf{L}_{\mathsf{trans}}$ over $\Lambda = \{\mathsf{L}_{\mathsf{wreg}}, \mathsf{L}_{\mathsf{Queue}}\}$ given in Fig. 1, and let us assume we were to show that I_{trans} is compositionally correct. Our aim here is only to outline the proof obligations that must be discharged; later in §5 we give a full proof in the more general weak memory setting.

1. For the first condition of compositional correctness, we must show I_{trans} preserves well-formedness: if the abstract history H is well-formed, then so is any corresponding concrete history $H' \in H \cdot I_{\mathsf{trans}}$. This is straightforward as the well-formedness conditions of $\mathsf{L}_{\mathsf{wreg}}$ and $\mathsf{L}_{\mathsf{Queue}}$ are trivial, and $\mathsf{L}_{\mathsf{trans}}$ does not use any existing tag.
2. For the second condition of compositional correctness, we must show that I_{trans} preserves consistency in the other direction: keeping the notations as above, assuming H' is consistent for Λ, then H is consistent as specified by $\mathsf{L}_{\mathsf{trans}}$. There are two parts to this obligation, as we also have to show that the $\mathsf{L}_{\mathsf{trans}}$'s operations tagged with T satisfy the global consistency predicate of the library.
3. The last condition holds vacuously as $\mathsf{L}_{\mathsf{trans}}$ does not use any existing tags.

Example 6 (A Client of $\mathsf{L}_{\mathsf{trans}}$). To see how the global consistency specifications work, consider a simple min-max counter library, $\mathsf{L}_{\mathsf{mmcnt}}$, tracking the maximal and minimal integer it has been given. The $\mathsf{L}_{\mathsf{mmcnt}}$ is to be used within

```
method mmNew() :=                    method mmMin(x) :=
 (PTNewReg(), PTNewReg())              PTRead(x.1)

method mmAdd(x, n) :=                method mmMax(x) :=
  PTWrite(min(n, PTRead(x.1)))         PTRead(x.2)
  PTWrite(max(n, PTRead(x.2)))
```

Fig. 2. Implementation I_{mmcnt} of $\mathsf{L}_{\mathsf{mmcnt}}$

$\mathsf{L}_{\mathsf{trans}}$ transactions, and provides four methods: mmNew() to construct a min-max counter, $\mathtt{mmAdd}(x, n)$, to add integer n to the min-max counter, and $\mathtt{mmMin}(x)$ and $\mathtt{mmMax}(x)$ to read the respective values.

We present the I_{mmcnt} implementation over $\mathsf{L}_{\mathsf{trans}}$ in Fig. 2. The idea is simply to track two integers denoting the minimal and maximal values of the numbers that have been added. Interestingly, even though they are stored in $\mathsf{L}_{\mathsf{trans}}$ registers, the implementation does not begin or end transactions: this is the responsibility of the client to avoid nesting transactions. This is enforced by $\mathsf{L}_{\mathsf{mmcnt}}$ using a global well-formedness predicate. Moreover, the mmAdd operation is tagged with T from the $\mathsf{L}_{\mathsf{trans}}$ library, ensuring that it behaves well w.r.t. transactions. A non-example is a version of I_{mmcnt} where the minimum is in a $\mathsf{L}_{\mathsf{trans}}$ register, but the max is in a "normal" $\mathsf{L}_{\mathsf{wreg}}$ register. This breaks the atomicity guarantee of transactions.

Formally, the interface L_{mmcnt} has four methods as above, where mmNew is the only constructor. The set of used tags is $\mathrm{TAGS}_{\mathrm{dep}} = \{\mathrm{T}, \mathrm{P}^{\mathsf{tr}}\}$, and all L_{mmcnt} methods are tagged with T as they all use primitives from $\mathsf{L}_{\mathsf{trans}}$. The consistency predicate is defined using the obvious sequential specification S_{mmcnt}, which states that calls to mmMin return the minimum of all integers previously given to mmAdd in the sequential history. We lift this to (concurrent) histories as follows. A history $H \in \mathbf{Hist}(\mathsf{L}_{\mathsf{mmcnt}})$ is in $\mathsf{L}_{\mathsf{mmcnt}}.\mathcal{S}_{\mathsf{c}}$ if there exists $E_\ell \in S_{\mathsf{mmcnt}}$ that is a $\prec$-linearization of $E_1[\mathrm{P}^{\mathsf{tr}}] \cdot E_2[\mathrm{P}^{\mathsf{tr}}] \cdots E_{n-1} \cdot E_n[\mathrm{P}^{\mathsf{tr}}]$, where H constructs n eras decomposed as $H = E_1 \cdot ⯇ \cdots ⯇ \cdot E_n$ (recall that $E[\mathrm{P}^{\mathsf{tr}}]$ denotes the sub-history with events tagged with P^{tr}, that is, persisted events.). The global specification and well-formedness conditions of $\mathsf{L}_{\mathsf{mmcnt}}$ are trivial. Because $\mathsf{L}_{\mathsf{mmcnt}}$ uses tag T of $\mathsf{L}_{\mathsf{trans}}$, a well-formed history of $\mathsf{L}_{\mathsf{mmcnt}}$ must satisfy $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{wf}}$, which requires that all operations tagged with T be inside transactions, and $\mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$ guarantees that $\mathsf{L}_{\mathsf{mmcnt}}$ operations persist atomically in a transaction.

When proving that the implementation in Figure 2 satisfies $\mathsf{L}_{\mathsf{mmcnt}}$ using compositional correctness, one proof obligation is to show that, given histories $H \in \mathbf{Hist}(\{\mathsf{L}_{\mathsf{trans}}, \mathsf{L}_{\mathsf{mmcnt}}, \star_{\mathsf{L}_{\mathsf{trans}}}\})$ and $H' \in H \cdot I_{\mathsf{mmcnt}} \subseteq \mathbf{Hist}(\{\mathsf{L}_{\mathsf{trans}}, \star_{\mathsf{L}_{\mathsf{trans}}}\})$, if $\pi_{\mathsf{L}_{\mathsf{trans}}}(H') \in \mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$, then $\pi_{\mathsf{L}_{\mathsf{trans}}}(H) \in \mathsf{L}_{\mathsf{trans}}.\mathcal{T}_{\mathsf{c}}$. This corresponds precisely to the fact that min-max counter operations persist atomically in a transaction, assuming the primitives it uses do as well.

2.9 Generic Durable Persistency Theorems

We consider another family of libraries with persistent reads/writes guaranteeing the following:

> if one replaces regular (volatile) reads/writes in a *linearizable* implementation with persistent ones, then the implementation obtained is *durably linearizable.*

We consider two such such libraries: FliT [35] and Mirror [10]. Thanks to our framework, we formalise the statement above for the first time and prove it for both Flit and Mirror against a realistic consistency (concurrency) model (see §4).

3 Generalization to weak-memory

This section sketches how we generalize the framework presented in the previous section to the weak memory, where events generated by the program are not totally ordered. For lack of space, the technical details, which largely follow that of the previous section, are relegated to the Appendix [34]. The purpose of this section is to give an idea of how *executions*, a standard tool in the semantics of weak memory, generalize the *histories* we used in the Overview section, and to give enough context for the case studies that follow.

Unlike the histories that we discussed in the previous section, in which events are totally ordered by a notion of time, events in executions are only partially ordered, reflecting that instructions executed in parallel are not naturally ordered. Formally, an execution is thus a set of events equipped with a partial order which represents the ordering between events from the same thread. This partial order, written po, for program-order, is depicted with black arrows in Fig. 3, where it orders minimally the initial event, and the two events of each thread according to the source code. Additional edges indicate, for each read-event returning the value v, the write-event that provided the value v: in that case, an rf-edge from the write-event to the read-event is added to the execution.

$$\begin{tikzcd}
 & {[init]} \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend left, "\mathrm{rf}"] & \\
R(x){:}5 \arrow[d, "\mathrm{po}"'] & & R(y){:}0 \arrow[d, "\mathrm{po}"] \\
W(y,2){:}() & & W(x,5){:}() \arrow[ull, dashed, "\mathrm{rf}"']
\end{tikzcd}$$

Fig. 3. An execution of the program P:
$a = x;\ y = 2 \parallel a = y;\ x = 5$

To be able to reason about synchronization, the notion of happens-before needs to be adapted to this setting. It is defined using po and an additional type of edge, *synchronizes-with*, written sw, which denotes that two events synchronize with each other, and in particular that one happens before the other. Usually, sw $\subseteq$ rf, for example between a release-write and an acquire-read in the C11 memory model. Given these sw edges, the happens-before order they induce, which generalizes $\prec$ from the previous section is defined as the transitive closure $(\mathrm{po} \cup \mathrm{sw})^+$. This is not sufficient however, because we consider *partial* executions G where the focus is on a subset of the libraries in some unknown global execution G', that is: $G = G' \downharpoonleft \mathsf{L}$. Therefore, external events (in G' but not in G) may induce happens-before relations between events of G, yet we want to specify library L without referring to any such execution G' that contains it. To solve this issue, we use the technique of [26], and we add a final type of edge to executions: hb, which corresponds to both the external and the internal synchronization. Because of the latter, it must contain the internal synchronization: $\mathrm{po} \cup \mathrm{sw} \subseteq \mathrm{hb}$.

To summarize, an execution is a tuple $\langle E, \mathrm{po}, \mathrm{rf}, \mathrm{sw}, \mathrm{hb}\rangle$ comprised of a set E of events, and of the relations we just described. A library specification is the same as in the previous section, *mutatis mutandis*. The sets of executions that are parts of specifications are defined using a formalism developed in the weak memory model literature. A set $\mathcal{S}$ of executions is described with conditions about relations built from po, rf, etc. Given a set V of events, we denote by $[V]$ the relation $V \times V$, and we denote by $R_1; R_2$ the standard composition of two relations R_1 and R_2. For example, if R denotes the set of read-events of an execution and W the set of write-events, the condition $[W]; \mathrm{rf}; [R] \subseteq \mathrm{sw}$ states that if there is a rf-edges between two events $e_1 \in W$ and $e_2 \in R$ of an execution, there must also be a sw synchronization edge between e_1 and e_2.

As in the previous section, the tag system allows the library specification to state which events must have been persisted in a valid execution. The semantics of a program is a set of executions that contain events from all the libraries used by the program; and whose happens-before order satisfy $\mathrm{hb} = (\mathrm{po} \cup \mathrm{sw})^+$, as there are no external synchronization in the executions of the whole program. The Appendix [34] details how our framework is defined in this more general setting.

4 Case Study: Durable Linearizability with FliT and Mirror

We consider a family of libraries that provide a simple interface with persistent memory accesses (reads and writes), allowing one to convert any linearisable implementation to a durably linearisable one by replacing regular (volatile) accesses with persistent ones supplied by the library. Specifically, we consider two such libraries FliT [35] and Mirror [10]; we specify them both in our framework, prove their implementations sound against their respective specifications, and further prove their general result for converting data structures.

4.1 The FliT Library

FliT [35] is a persistent library that provides a simple interface very close to Px86, but with stronger persistency guarantees, which make it easier to implement durable data structures. Specifically, a FliT object ℓ can be accessed via

```
method $\texttt{wr}_\pi(\ell, v)$ :
  if $\pi = \mathsf{p}$ then
    fetch-and-add($\textit{flit-counter}(\ell), 1$);
    write($\ell, v$);
    $\mathsf{flush_{opt}}(\ell)$;
    fetch-and-add($\textit{flit-counter}(\ell), -1$);
  else
    sfence;
    write($\ell, v$);

method $\texttt{rd}_\pi(\ell)$ :
  local $v$ = read($\ell$);
  if $\pi = \mathsf{p} \land \textit{flit-counter}(\ell) > 0$ then
    $\mathsf{flush_{opt}}(\ell)$;
  return $v$;

method finishOp :
  sfence;
```

Fig. 4. FliT library implementation in Px86

write and read methods, $\texttt{wr}_\pi(\ell, v)$ and $\texttt{rd}_\pi(\ell)$, as well as standard read-modify-write methods. Each write (resp. read) operation has two variants, denoted by the *type* $\pi \in \{\mathsf{p}, \mathsf{v}\}$. This type specifies if the write (resp. read) is *persistent* ($\pi = \mathsf{p}$) in that its effects must be persisted, or *volatile* ($\pi = \mathsf{v}$) in that its persistency has been optimised and offers weaker guarantees. The default access type is persistent (p), and the volatile accesses may be used as optimizations when weaker guarantees suffice. Wei et al. [35] introduce a notion of *dependency* between different operations as follows. If a (persistent or volatile) write w depends on a persistent write w', then w' persists before w. If a persistent read r reads from a persistent write w, then r depends on w and thus w must be persisted upon reading if it has not already persisted. Though simple, FliT provides a strong guarantee as captured by a general result for correctly converting volatile data structures to persistent ones: if one replaces every memory access in the implementation of a *linearizable* data-structure with the corresponding persistent FliT access, then the resulting data structure is *durably linearizable.*

Compared to the original FliT development, our soundness proof is more formal and detailed: it is established against a formal specification (rather than an English description) and with respect to the formal Px86 model.

FliT Interface. The FliT interface uses the $\mathrm{P}^{\mathsf{Px86}}$ from Px86 and contains a single constructor, new, allocating a new FliT location, as well as three other methods below, the last two of which are durable:

- $\texttt{rd}_\pi(\ell)$ with $\pi \in \{\mathsf{p}, \mathsf{v}\}$, for a π-read from ℓ;
- $\texttt{wr}_\pi(\ell, v)$ with $\pi \in \{\mathsf{p}, \mathsf{v}\}$, denoting a π-write of value $v \in \mathbf{Val}$ to ℓ; and
- finishOp, which waits for previously executed operations to persist.

We write R and W respectively for the read and write events, and add the superscript π (e.g. R^{p}) to denote such events with the given persistency mode.

FliT Specification. We develop a formal specification of FliT in our framework, based on its original informal description. The correctness of FliT executions is described via a *dependency* relation that contains the program order and the total execution (linearization) order restricted to persistent write-read operations on the same location. Note that this dependency notion is stronger than the customary definitions that use a rf relation (as in the Px86 specification) instead of lin, because a persistent read may not read directly from a persistent write w, but rather from another later (lin-after w) write.

Definition 19 (FliT execution Correctness). *A FliT execution $\mathcal{G}$ is correct if there exists a 'reads-from' relation* rf *and a total order* lin $\supseteq \mathcal{G}.\mathrm{hb}$ *on* $\mathcal{G}.E$ *and an order* nvo *such that:*

1. *Each read event reads from the most recent previous write to the same location:*
   $\mathrm{rf} = \bigcup_{\ell \in \mathbf{Loc}}([W_\ell]; \mathrm{lin}; [R_\ell]) \setminus (\mathrm{lin}; [W_\ell]; \mathrm{lin})$
2. *Reads return the value written by the write they read from:*
   $(w, r) \in \mathrm{rf} \Rightarrow \exists \ell, \pi, \pi', v.\ \mathrm{lab}(r) = \mathtt{rd}_{\pi'}(\ell) : v \wedge \mathrm{lab}(w) = \mathtt{wr}_{\pi}(\ell, v) : -$
3. *Persistent writes persist before every other later dependent write:*
   $[W^{\mathsf{p}}]; (\mathrm{po} \cup \bigcup_{\ell \in \mathbf{Loc}} [W^{\mathsf{p}}_\ell]; \mathrm{lin}; [R^{\mathsf{p}}_\ell])^+; [W] \subseteq \mathrm{nvo}$
4. *Persistent writes before a* `finishOp` *persist:*
   $dom([W^{\mathsf{p}}]; (\mathrm{po} \cup \bigcup_{\ell \in \mathbf{Loc}} [W^{\mathsf{p}}_\ell]; \mathrm{lin}; [R^{\mathsf{p}}_\ell])^+; [\mathit{finishOp}]) \subseteq \lfloor \mathrm{P}^{\mathit{Px86}} \rfloor$
5. *And* nvo *is a persist order:* $dom(\mathrm{nvo}; \lfloor Ptag \rfloor) \subseteq \lfloor Ptag \rfloor$.

Px86 implementation of FliT. The implementation of FliT methods is given in Fig. 4. Whereas a naive implementation of this interface would have to issue a flush instruction both after persistent writes and in persistent reads, the implementation shown associates each location with a counter to avoid performing superfluous flushes when reading from a location whose value has already persisted. Specifically, a persistent write on ℓ increments its counter before writing to and flushing it, and decrements the counter afterwards. As such, persistent reads only need to issue a flush if the counter is positive (i.e. if there is a concurrent write that has not executed its flush yet).

Theorem 2. *The implementation of FliT in Fig. 4 is correct.*

FliT and Durable Linearizability. Given a data structure implementation I, let $p(I)$ denote the implementation obtained from I by 1) replacing reads/writes in the implementation with their corresponding persistent FliT instructions, and 2) adding a call to `finishOp` right before the end of each method. We then show that given an implementation I, if I is linearizable, then $p(I)$ is *durably linearizable*[3]. We assume that all method implementations are single-threaded, i.e. all plain executions $I(m(\boldsymbol{v}))$ are totally ordered.

Theorem 3. *If* $\mathit{Px86} \vDash I : \mathit{Lin}(S)$, *then* $\mathit{FliT} \vDash p(I) : \mathit{DurLin}(S)$.

4.2 The Mirror Library

The Mirror [10] persistent library has similar goals to FliT. The main difference between the two is that Mirror operations do not offer two variants, and their operations are implemented differently from those of FliT. Specifically, in Mirror each location has two copies: one in persistent memory to ensure durability,

[3] The definition here is the same as in §2, as hb-linearizations of the execution still yield sequential executions.

and one in volatile memory for fast access. As such, read operations are implemented as simple loads from volatile memory, while writes have a more involved implementation than those of FliT.

We present the Mirror specification and implementation in the technical appendix where we also prove that its implementation is correct against its specification. As with FliT, we further prove that Mirror can be used to convert linearizable data structures to durably linearizable ones, as described above.

5 Case Study: Persistent Transactional Library

We revisit the $\mathsf{L_{trans}}$ transactional library, develop its formal specification and verify its implementation (Fig. 1) against it. Recall the simple $\mathsf{L_{trans}}$ implementation in Fig. 1 and that we do not allow for nested transactions. The implementation uses an *undo-log* which records the former values of persistent registers (locations) modified in a transaction. If, after a crash, the recovery mechanism detects a partially persisted transaction (i.e. the last entry in the undo log is not COMMITTED), then it can use the undo-log to restore registers to their former values. The implementation uses a durably linearizable queue library[4] Q, and assumes that it is *externally synchronized*: the user is responsible for ensuring no two transactions are executed in parallel. We formalize this using a global well-formedness condition.

Later in §5.2 we develop a wrapper library $\mathsf{L_{Strans}}$ for $\mathsf{L_{trans}}$ that additionally provides synchronization using locks and prove that our implementation of this library is correct. To do this, we need to make small modifications to the structure of the specification: the specification in §2 requires that any 'transaction-aware operation' (i.e. those tagged with T) be enclosed in calls to `PTBegin` and `PTEnd`. Since $\mathsf{L_{Strans}}$ wraps the calls to `PTBegin` and `PTEnd`, the well-formedness condition needs to be generalized to allow operations tagged with T to appear between calls to operations that behave like `PTBegin` and `PTEnd`. To that end, we add two new tags B and E to denote such operations, respectively.

5.1 Specification

The $\mathsf{L_{trans}}$ library provides four *tags*: 1) T for transaction-aware 'client' operations; 2) P^{tr} for operations that have persisted using transactions; and 3) B, E for operations that begin and end transactions, respectively. We write $\mathcal{R}, \mathcal{W}, \mathcal{B}, \mathcal{E}, \mathcal{RC}$ respectively for the sets of events labeled with read, write, begin, end and recovery methods. As before, we write e.g. $\lfloor \text{T} \rfloor$ for the set of events tagged with T. Note that while $\mathcal{B}$ denotes the set of the begin events in library $\mathsf{L_{trans}}$, the $\lfloor \text{B} \rfloor$ denotes the set of all events that are tagged with B, which includes $\mathcal{B}$ (of library $\mathsf{L_{trans}}$) as well as events of *other* (non-$\mathsf{L_{trans}}$) libraries that may be tagged with B; similarly for $\mathcal{E}$ and $\lfloor \text{E} \rfloor$. As such, our local specifications below (i.e. local well-formedness

[4] For example, take any linearizable queue implementation and use the FliT library as described in §4.

and consistency) are defined in terms of $\mathcal{B}$ and $\mathcal{E}$, whereas our global specifications are defined in terms of $\lfloor\mathrm{B}\rfloor$ and $\lfloor\mathrm{E}\rfloor$. As before, for brevity we write e.g. $[\mathrm{T}]$ as a shorthand for the relation $[\lfloor\mathrm{T}\rfloor]$. We next define the 'same-transaction' relation strans:

$$\mathrm{strans} \triangleq [\lfloor\mathrm{B}\rfloor \cup \lfloor\mathrm{E}\rfloor \cup \lfloor\mathrm{T}\rfloor]; (\mathrm{po} \cup \mathrm{po}^{-1}); [\lfloor\mathrm{B}\rfloor \cup \lfloor\mathrm{E}\rfloor \cup \lfloor\mathrm{T}\rfloor] \setminus ((\mathrm{po}; [\mathrm{E}]; \mathrm{po}) \cup (\mathrm{po}; [\mathrm{B}]; \mathrm{po}))$$

An execution is locally well-formed iff the following hold:

1. A transaction must be opened before it is closed: $\mathcal{E} \subseteq \mathit{rng}([\mathcal{B}]; \mathrm{po})$
2. Transactions are not nested and are matching: $[\mathcal{E}]; \mathrm{po}; [\mathcal{E}] \subseteq [\mathcal{E}]; \mathrm{po}; [\mathcal{B}]; \mathrm{po}; [\mathcal{E}]$ and $[\mathcal{B}]; \mathrm{po}; [\mathcal{B}] \subseteq [\mathcal{B}]; \mathrm{po}; [\mathcal{E}]; \mathrm{po}; [\mathcal{B}]$
3. Transactions must be externally synchronized: $\mathcal{E} \times \mathcal{B} \subseteq \mathrm{hb} \cup \mathrm{hb}^{-1}$
4. The recovery routine must be called after each crash before using the library: $\lightning; \mathrm{hb}; \lfloor\mathrm{B}\rfloor \subseteq \lightning; \mathrm{hb}; [\mathcal{RC}]; \mathrm{hb}; \lfloor\mathrm{B}\rfloor$
5. Events are correctly tagged: $\mathcal{W} \cup \mathcal{R} \subseteq \lfloor\mathrm{T}\rfloor$

An execution is globally well-formed if client operations are inside transactions:

6. $\lfloor\mathrm{T}\rfloor \subseteq \mathit{rng}([\mathrm{B}]; \mathrm{po})$
7. $[\mathrm{E}]; \mathrm{po}; [\mathrm{T}] \subseteq [\mathrm{E}]; \mathrm{po}; [\mathrm{B}]; \mathrm{po}; [\mathrm{T}]$

An execution is locally-consistent if there exists a relation rf satisfying:

8. rf relates writes to reads, $\mathrm{rf} \subseteq \mathcal{W} \times \mathcal{R}$, such that each read is related to exactly one write (i.e. rf^{-1} is total and functional).
9. Reads access the most recent write: $\mathrm{rf}^{-1}; \mathrm{hb} \subseteq \mathrm{hb}$
10. External reads (reading from a different transaction) read from persisted writes: $\mathit{dom}(\mathrm{rf} \setminus \mathrm{strans}) \subseteq \lfloor\mathrm{P}^{\mathsf{tr}}\rfloor$

An execution is globally-consistent if there exists an order nvo over $\lfloor\mathrm{T}\rfloor$ satisfying:

11. Transactions are nvo-ordered: $[\mathrm{E}]; \mathrm{hb}; [\mathrm{B}] \subseteq \mathrm{nvo}$
12. nvo is the persistance order: $\mathit{dom}(\mathrm{nvo}; [\mathrm{P}^{\mathsf{tr}}]) \subseteq \lfloor\mathrm{P}^{\mathsf{tr}}\rfloor$;
13. Either all the events or none of the events in a transaction persist (atomicity): $[\mathrm{P}^{\mathsf{tr}}]; \mathrm{strans}; [\mathrm{T}] \subseteq [\mathrm{P}^{\mathsf{tr}}]$
14. All events of a completed transaction (ones with an associated end event) persist: $\lfloor\mathrm{E}\rfloor^{c} \subseteq \lfloor\mathrm{P}^{\mathsf{tr}}\rfloor$, where $\lfloor\mathrm{E}\rfloor^{c}$ denotes the set of method calls tagged with E which have completed.

Theorem 4. *The L_{trans} implementation in Fig. 1 over Px86 is correct.*

5.2 Vertical Library Composition: Adding Internal Synchronization

We next demonstrate how our framework can be used for *vertical library composition*, where an implementation of one library comprises calls to other libraries with non-trivial global specifications. To this end, we develop $\mathsf{L}_{\mathsf{Strans}}$, a wrapper library around $\mathsf{L}_{\mathsf{trans}}$ that is meant to be simpler to use by providing synchronization internally: rather than the user ensuring synchronization for $\mathsf{L}_{\mathsf{trans}}$, one can use $\mathsf{L_{Strans}}$ to prevent two transactions from executing in parallel. More formally, the well-formedness condition (3) of $\mathsf{L_{trans}}$ becomes a correctness guarantee of $\mathsf{L_{Strans}}$. We consider a simple implementation of $\mathsf{L_{Strans}}$ that uses a global lock acquired at the beginning of each transaction and released at the end as shown below.

```
globals lock := L.new()        method LPTBegin() := L.acq(lock);PTBegin()
                               method LPTEnd() := PTEnd();L.rel(lock)
```

Theorem 5. *The implementation of* $\mathsf{L_{Strans}}$ *above is correct.*

Using compositional correctness, the main proof obligation is the condition stipulating that the implementation be well-formed, ensuring that $\mathsf{L_{trans}}$ is used correctly by the $\mathsf{L_{Strans}}$ implementation. This is straightforward as we can assume there exists an immediate prefix that is consistent. The existence of the hb-ordering of calls to `PTBegin` and `PTEnd` follows from the consistency of the global lock used by the implementation.

5.3 Horizontal Library Composition

We next demonstrate how our framework can be used for *horizontal library composition*, where a *client* program comprises calls to multiple libraries. To this end, we develop a simple library, $\mathsf{L_{cntr}}$, providing a persistent counter to be used in *sequential* (single-threaded) settings: If a client uses $\mathsf{L_{cntr}}$ in concurrent settings, it must call its methods within critical sections. The $\mathsf{L_{cntr}}$ provides three operations to create (NewCounter), increment (CounterInc) and read a counter (CounterRead). The specification and implementation of $\mathsf{L_{cntr}}$ are given in [34]

As $\mathsf{L_{cntr}}$ uses the tags of $\mathsf{L_{trans}}$, we define $\mathsf{L_{cntr}}.\Lambda_{\text{tags}} \triangleq \{\mathsf{L_{trans}}\}$. The all the operations are tagged with T. As such, $\mathsf{L_{cntr}}$ inherits the global well-formedness condition of $\mathsf{L_{trans}}$, meaning that $\mathsf{L_{cntr}}$ operations must be used within transactions (i.e. hb-between operations respectively tagged with B and E). Putting it all together, the following client code snippet uses $\mathsf{L_{cntr}}$ in a correct way, even though $\mathsf{L_{cntr}}$ has no knowledge of the existence of $\mathsf{L_{Strans}}$.

```
c = NewCounter(); LPTBegin(); CounterInc(c);  CounterInc(c);  LPTEnd();
```

Specifically, the above is an instance of horizontal library composition (as the client comprises calls to both $\mathsf{L_{Strans}}$ and $\mathsf{L_{cntr}}$), facilitated in our framework through global specifications.

6 Conclusions, Related and Future Work

We presented a framework for specifying and verifying persistent libraries, and demonstrated its utility and generality by encoding existing correctness notions within it and proving the correctness of the FliT and Mirror libraries, as well as a persistent transactional library.

Related Work. The most closely related body of work to ours is [26]. However, while their framework can be used for specifying only the consistency guarantees of a library, ours can be used to specify both consistency and persistency guarantees. More generally, our tag system extends the expressivity of [26] with support for *global* effects such as some types of fences.

Existing literature includes several works on formal persistency models, both for hardware [25,30,31,5,6,19,29,28] and software [4,21,11], as well as correctness conditions for persistent libraries such as durable linearizability [17]. As we showed in §3, such models can be specified in our framework.

There have been works [33] to specify libraries using an operational approach instead of the declarative approach that we advocate for here. While it is not generic in the memory model, it support weak memory, with a fragment of the C++ 11 memory model, and supports synchronization that is internal and external to the library. Another framework for formalizing behavior of concurrent objects in the presence of weak memory is [18], which is more syntactic as our framework: they use a process calculus, which allows them to handle callbacks between the library and the client. Extending our framework, which is more semantic, to handle this setting would probably require shifting from executions/histories to something similar to game semantics.

Additionally, there are several works on implementing and verifying algorithms that operate on NVM. [9] and [36] respectively developed persistent queue and set implementations in Px86. [8] provided a formal correctness proof of the implementation in [36]. All three of [8,36,9] assume that the underlying concurrency model is SC [23], rather than that of Px86 (namely TSO). As we demonstrated in §4–§5 we can use our framework to verify persistent implementations *modularly* while remaining faithful to the underlying concurrency model. [27,2] have developed persistent program logics for verifying programs under Px86. [20] recently formalized the consistency and persistency semantics of the Linux ext4 file system, and developed a model-checking algorithm and tool for verifying the consistency and persistency behaviors of ext4 applications such as text editors.

Recently, and independently to this work, Bodenmüller *et al* [3] have proved the correctness of the Flit library under TSO. They used an operational approach, and modeled the libraries and the memory and persistency models operationally using automata, and proved a simulation result using KIV a specialized proof assistant. As for this paper, they proved that a linearizable library using Flit becomes durably linearizable.

Future Work. We believe our framework will pave the way for further work on verifying persistent libraries, whether manually (as done here), possibly with the assistance of an interactive theorem prover and/or program logics such as those of [7,27,2], or automatically via model checking. The work of [7] uses the framework of [26] to specify data structures in a program logic, and it would be natural to extend it to our framework for persistency. Existing work in the latter research direction, e.g. [12,20], has so far only considered low-level properties, such as the absence of races or the preservation of user-supplied invariants. It has not yet considered higher-level functional correctness properties, such as durable linearizability and its variants. We believe our framework will be helpful in that regard. In a more theoretical direction, it would be interesting to understand how our compositional correctness theorem fits in general settings for abstract logical relations such as [16].

Acknowledgments

We thank the anonymous reviewers for their feedback. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349). Raad is funded by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

References

1. Aguilera, M.K., Frolund, S.: Strict linearizability and the power of aborting. Tech. Rep. HPL-2003-241 (2013)
2. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based owicki–gries reasoning for persistent x86-tso. In: Sergey, I. (ed.) Programming Languages and Systems. pp. 234–261. Springer International Publishing, Cham (2022)
3. Bodenmüller, S., Derrick, J., Dongol, B., Schellhorn, G., Wehrheim, H.: A fully verified persistency library. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 26–47. Springer Nature Switzerland, Cham (2024)
4. Chakrabarti, D.R., Boehm, H.J., Bhandari, K.: Atlas: Leveraging locks for non-volatile memory consistency. SIGPLAN Not. **49**(10), 433–452 (Oct 2014). `https://doi.org/10.1145/2714064.2660224`, `http://doi.acm.org/10.1145/2714064.2660224`
5. Cho, K., Lee, S.H., Raad, A., Kang, J.: Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-X86 and Armv8. p. 16–31. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021)
6. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 133–146. SOSP '09, ACM, New York, NY, USA (2009). `https://doi.org/10.1145/1629575.1629589`, `http://doi.acm.org/10.1145/1629575.1629589`
7. Dang, H.H., Jung, J., Choi, J., Nguyen, D.T., Mansky, W., Kang, J., Dreyer, D.: Compass: Strong and compositional library specifications in relaxed memory separation logic. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022 (2022)
8. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods – The Next 30 Years. pp. 179–195. Springer International Publishing, Cham (2019)
9. Friedman, M., Herlihy, M., Marathe, V., Petrank, E.: A persistent lock-free queue for non-volatile memory. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 28–40. PPoPP '18, Association for Computing Machinery, New York, NY, USA (2018). `https://doi.org/10.1145/3178487.3178490`, `https://doi.org/10.1145/3178487.3178490`

10. Friedman, M., Petrank, E., Ramalhete, P.: Mirror: making lock-free data structures persistent. In: Freund, S.N., Yahav, E. (eds.) PLDI '21. pp. 1218–1232 (2021)
11. Gogte, V., Diestelhorst, S., Wang, W., Narayanasamy, S., Chen, P.M., Wenisch, T.F.: Persistency for synchronization-free regions. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 46–61. PLDI 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3192366.3192367, http://doi.acm.org/10.1145/3192366.3192367
12. Gorjiara, H., Xu, G.H., Demsky, B.: Yashme: Detecting persistency races. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 830–845. ASPLOS 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3503222.3507766, https://doi.org/10.1145/3503222.3507766
13. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: POPL (2015)
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
15. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
16. Hermida, C., Reddy, U.S., Robinson, E.P.: Logical relations and parametricity – a reynolds programme for category theory and programming languages. Electronic Notes in Theoretical Computer Science **303**, 149–180 (2014), proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013)
17. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille, C., Ilcinkas, D. (eds.) DISC. Lecture Notes in Computer Science, vol. 9888, pp. 313–327 (2016)
18. Jagadeesan, R., Petri, G., Pitcher, C., Riely, J.: Quarantining weakness. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 492–511. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
19. Khyzha, A., Lahav, O.: Taming x86-tso persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
20. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: Persistency semantics for verification under ext4. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434324, https://doi.org/10.1145/3434324
21. Kolli, A., Gogte, V., Saidi, A., Diestelhorst, S., Chen, P.M., Narayanasamy, S., Wenisch, T.F.: Language-level persistency. In: Proceedings of the 44th Annual International Symposium on Computer Architecture. pp. 481–493. ISCA '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3079856.3080229, http://doi.acm.org/10.1145/3079856.3080229
22. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. SIGPLAN Not. **52**(6), 618–632 (jun 2017). https://doi.org/10.1145/3140587.3062352, https://doi.org/10.1145/3140587.3062352
23. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (Sep 1979). https://doi.org/10.1109/TC.1979.1675439, http://dx.doi.org/10.1109/TC.1979.1675439
24. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. pp. 265–276. ISCA '14, IEEE Press, Piscataway, NJ, USA (2014), http://dl.acm.org/citation.cfm?id=2665671.2665712

25. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. p. 265–276. ISCA '14, IEEE Press (2014)
26. Raad, A., Doko, M., Rožić, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. POPL (2019)
27. Raad, A., Lahav, O., Vafeiadis, V.: Persistent owicki-gries reasoning: A program logic for reasoning about persistent programs on intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). `https://doi.org/10.1145/3428219`, `https://doi.org/10.1145/3428219`
28. Raad, A., Maranget, L., Vafeiadis, V.: Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022)
29. Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. Proc. ACM Program. Lang. **2**(OOPSLA), 137:1–137:27 (Oct 2018). `https://doi.org/10.1145/3276507`, `http://doi.acm.org/10.1145/3276507`
30. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1–11:31 (2020)
31. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1–135:27 (Oct 2019)
32. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (Jul 2010). `https://doi.org/10.1145/1785414.1785443`, `http://doi.acm.org/10.1145/1785414.1785443`
33. Singh, A.K., Lahav, O.: An operational approach to library abstraction under relaxed memory concurrency. Proc. ACM Program. Lang. **7**(POPL) (jan 2023). `https://doi.org/10.1145/3571246`, `https://doi.org/10.1145/3571246`
34. Stefanesco, L., Raad, A., Vafeiadis, V.: Specifying and verifying persistent libraries (with appendix). CoRR **abs/2306.01614** (2023). `https://doi.org/10.48550/ARXIV.2306.01614`, `https://doi.org/10.48550/arXiv.2306.01614`
35. Wei, Y., Ben-David, N., Friedman, M., Blelloch, G.E., Petrank, E.: Flit: a library for simple and efficient persistent algorithms. In: Lee, J., Agrawal, K., Spear, M.F. (eds.) PPoPP '22. pp. 309–321 (2022)
36. Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., Petrank, E.: Efficient lock-free durable sets. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). `https://doi.org/10.1145/3360554`, `https://doi.org/10.1145/3360554`

Hyperproperty Verification as CHC Satisfiability

Shachar Itzhaky[1](✉), Sharon Shoham[2], and Yakir Vizel[1]

[1] Technion, Haifa, Israel
shachari@cs.technion.ac.il
[2] Tel-Aviv University, Tel Aviv-Yafo, Israel

Abstract. Hyperproperties specify the behavior of a system across multiple executions, and are an important extension of regular temporal properties. So far, such properties have resisted comprehensive treatment by software model-checking approaches such as IC3/PDR, due to the need to find not only an inductive invariant but also a *total* alignment of different executions that facilitates simpler inductive invariants.
We show how this treatment is achieved via a reduction from the verification problem of $\forall^*\exists^*$ hyperproperties to Constrained Horn Clauses (CHCs). Our starting point is a set of universally quantified formulas in first-order logic (modulo theories) that encode the verification of $\forall^*\exists^*$ hyperproperties over infinite-state transition systems. The first-order encoding uses uninterpreted predicates to capture the (1) witness function for existential quantification over traces, (2) alignment of executions, and (3) corresponding inductive invariant. Such an encoding was previously proposed for k-safety properties. Unfortunately, finding a satisfying model for the resulting first-order formulas is beyond reach for modern first-order satisfiability solvers. Previous works tackled this obstacle by developing specialized solvers for the aforementioned first-order formulas. In contrast, we show that the same problems can be encoded as CHCs and solved by existing CHC solvers. CHC solvers take advantage of the unique structure of CHC formulas and handle the combination of quantifiers with theories and uninterpreted predicates more efficiently.
Our key technical contribution is a logical transformation of the aforementioned sets of first-order formulas to equi-satisfiable sets of CHCs. The transformation to CHCs is sound and complete, and applying it to the first-order formulas that encode verification of hyperproperties leads to a CHC encoding of these problems. We implemented the CHC encoding in a prototype tool and show that, using existing CHC solvers for solving the CHCs, the approach already outperforms state-of-the-art tools for hyperproperty verification by orders of magnitude.

1 Introduction

Hyperproperties [15] are properties that relate multiple execution traces, either taken from a single program or from multiple programs. Checking such properties is known as *relational verification*, and is essential when reasoning about security policies, program equivalence, concurrency protocols, etc. Existing specification languages for hyperproperties [14,6,43] extend standard ones, e.g., temporal logic or Hoare logic, with (explicit or implicit) quantification over traces. This shifts

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 212–241, 2024.
https://doi.org/10.1007/978-3-031-57267-8_9

the focus from properties of individual traces to properties of *sets* of traces. For example, k-safety [15] is a class of hyperproperties, where k universal quantifiers are used to define a relational invariant over states originating from k traces.

This paper addresses verification of hyperproperties with $\forall^*\exists^*$ quantification over traces and a body of the form $\Box\phi$ (where $\Box$ stands for "globally"). This fragment captures many hypersafety (e.g., the aforementioned k-safety) and hyperliveness properties, and was shown by [8] to express a wide class of properties of interest, including generalized non-interference (GNI) [38].

Verification of hyperproperties is more challenging than verification of single-trace properties, and, as a result, has gained a lot of attention in recent years. Unlike single-trace properties, verification of properties of k traces requires the discovery of *relational* inductive invariants, which define the relation between states of k execution traces. Since the construction of invariants that hold between *any* k reachable states is hard (or even impossible, depending on the assertion logic), proving hyperproperties often hinges on finding an *alignment* of any k traces such that the invariant only needs to describe aligned states.

In the case of k-safety properties, an alignment of traces is often given by a *self composition* [5,44] of the program, composing different copies of the program (or several different programs) together, *e.g.*, by running the different copies in lockstep [48] or by more sophisticated composition schemes, *e.g.*, [24]. While self composition allows to reduce k-safety verification to standard safety verification, this reduction requires to choose the alignment of the different copies a-priori. The choice of alignment, however, has a significant effect on the complexity of the inductive invariants themselves, as demonstrated by [41]. This renders the standard reduction from k-safety verification to safety verification, based on a fixed alignment, impractical in many cases. As a result, finding a good alignment as part of relational verification has been a topic of interest in recent years [43,27,45,6,8].

In the case of hyperliveness properties that stem from the use of existential quantification over traces (*i.e.* $\forall^*\exists^*$ properties), complexity rises further. Verifying such hyperliveness properties calls for finding "witness" traces that match the universally quantified traces, in addition to the relational invariant and alignment. This reduces verification of $\forall^*\exists^*$ properties to the problem of inferring three ingredients: (i) a witness function for existential quantification over traces, (ii) an alignment of traces, and (iii) a corresponding relational inductive invariant. These ingredients are all interdependent: different witnesses call for different alignments and give rise to different invariants, with different levels of complexity. It is therefore desirable to search for the *combination* of the three of them *simultaneously*, which is the focus of this paper.

We propose a novel reduction from verification of hyperproperties with a $\forall^*\exists^*$ quantification prefix over infinite-state transition systems to satisfiability of Constrained Horn Clauses (CHCs) [11,10], also known as CHC-SAT. Importantly, the reduction does not fix any of the aforementioned verification ingredients, in particular, the alignment, a-priori. Instead, it is based on a CHC encoding of their joint requirements. The unique structure of CHCs makes it possible to adopt software model checking techniques (e.g. interpolation [39], IC3/PDR [32,35]) for solving them. Our reduction, thus, allows to use state-of-the-art CHC solvers [28,33,31,49] to achieve a highly efficient hyperproperty verification procedure.

While it is known that safety verification can be reduced to CHC-SAT, we are the first to show how inferring the combination of a witness function, a trace alignment and an inductive invariant for hyperproperties of the $\forall^*\exists^*$-fragment can be reduced to CHC-SAT.

The first step of our reduction to CHC-SAT is an encoding of the joint requirements of the witness-alignment-invariant ingredients as a set of universally quantified formulas in first-order logic (FOL) modulo theories, where uninterpreted predicates capture the witness, alignment and invariant, and first-order theories (*e.g.*, arithmetic and arrays) are used for modeling the transition system and the requirements. Such an encoding has been proposed by [41] for the problem of finding an invariant together with an alignment in the context of verification of k-safety properties (the universally quantified subset of this fragment). We extend their FOL encoding to $\forall^*\exists^*$ properties, based on the game semantics introduced in [8].

Unfortunately, the resulting FOL formulas are beyond what modern first-order satisfiability solvers can handle due to a combination of quantifiers with theories and uninterpreted predicates. In particular, the FOL formulas are not in the form of CHCs. As a result, previous works [41,45] that used a similar encoding could not rely on a (single) CHC-SAT query to find the alignment and invariant simultaneously. Instead, [41] resorted to an enumeration of potential alignments, using a separate CHC-SAT query to search for an inductive invariant (in a restricted language) for each candidate alignment. [45] developed a specialized solver that is able to handle these non-CHC formulas directly.

In contrast to previous works, we introduce a second step where we transform the set of universally quantified FOL formulas to a set of universally quantified CHCs. This step—which is also the key technical contribution of the paper—allows us to use any CHC solver for hyperproperty verification, and benefit from current and future developments in this lively area of research. We emphasize that the transformation to CHCs is surprising since it allows us to overcome a seemingly unavoidable obstacle: a disjunction of atomic formulas involving unknown predicates, which arises from the encoding of a choice between different alignment and witness options.

We implemented the reduction of $\forall^*\exists^*$-hyperproperty verification to CHC-SAT in a tool called HyHorn, on top of Z3 [23], using Spacer [31] as a CHC solver. Our results show that HyHorn is very efficient in verifying $\forall^*\exists^*$-hyperproperties, outperforming the state-of-the-art [45,8,41] by orders of magnitude.

Our main contributions are:

– We develop a satisfiability-preserving transformation of first-order formulas of a certain form to CHCs. The transformation is accompanied by a bi-directional translation of solutions.

```
pre(a₁ < a₂ ∧ b₁ > b₂)
squaresSum(int a, int b){
  assume(0 < a < b);
  int c=0;
  while (a<b) {c+=a*a; a++;}
  return c;
}
post(c₁ > c₂)
```

$$\boxed{\begin{array}{l} a_1 < a_2 \wedge b_1 > b_2 \rightarrow \\ \quad \forall \pi_1 : \neg(a < b), \pi_2 : \neg(a < b) \cdot \Box(c_1 > c_2) \end{array}}$$

(a)

$$\begin{array}{r} (1) \mathit{Init}(V_1) \wedge \mathit{Init}(V_2) \wedge a_2 > a_1 \wedge b_2 < b_1 \rightarrow \mathit{Inv}(V_1, V_2) \\ (2) \mathit{Inv}(V_1, V_2) \wedge A_{\{1\}}(V_1, V_2) \wedge \mathit{Tr}(V_1, V_1') \wedge V_2 = V_2' \rightarrow \mathit{Inv}(V_1', V_2') \\ (3) \mathit{Inv}(V_1, V_2) \wedge A_{\{2\}}(V_1, V_2) \wedge V_1 = V_1' \wedge \mathit{Tr}(V_2, V_2') \rightarrow \mathit{Inv}(V_1', V_2') \\ (4) \mathit{Inv}(V_1, V_2) \wedge A_{\{1,2\}}(V_1, V_2) \wedge \mathit{Tr}(V_1, V_1') \wedge \mathit{Tr}(V_2, V_2') \rightarrow \mathit{Inv}(V_1', V_2') \\ (5) \mathit{Inv}(V_1, V_2) \wedge A_{\{1\}}(V_1, V_2) \rightarrow a_1 < b_1 \\ (6) \mathit{Inv}(V_1, V_2) \wedge A_{\{2\}}(V_1, V_2) \rightarrow a_2 < b_2 \\ (7) \mathit{Inv}(V_1, V_2) \wedge A_{\{1,2\}}(V_1, V_2) \rightarrow (a_1 < b_1 \wedge a_2 < b_2) \\ \vee (a_1 \geq b_1 \wedge a_2 \geq b_2) \\ (8) \mathit{Inv}(V_1, V_2) \rightarrow ((a_1 \geq b_1 \wedge a_2 \geq b_2) \rightarrow c_1 > c_2) \\ (9) \mathit{Inv}(V_1, V_2) \rightarrow A_{\{1\}}(V_1, V_2) \vee A_{\{2\}}(V_1, V_2) \vee A_{\{1,2\}}(V_1, V_2) \end{array}$$

(b)

Fig. 1: (a) A program that computes the sum of squares of integer interval $[a, b)$ with a 2-safety specification for it, and (b) its first-order encoding.

- We apply the transformation to obtain, for the first time, a sound and complete reduction from verification of $\forall^*\exists^*$-OHyperLTL (w.r.t. a game semantics) to CHC-SAT. The reduction captures searching for an alignment, an $\exists^*$-witness function and an inductive invariant simultaneously. It is applicable to infinite-state transition systems, with the caveat that their branching degree needs to be finite (bounded by a constant) if the hyperproperty includes $\exists^*$ quantification.
- To handle $\exists^*$ in the presence of unbounded nondeterminism, we incorporate into the CHC encoding a sound abstraction based on a set of underapproximations ("restrictions").
- We implement a tool, HyHorn, that constructs CHCs for $\forall^*\exists^*$-OHyperLTL specifications, and solves them using SPACER. In most cases, HyHorn discovers the solution completely automatically, while in some, it uses predicate abstraction, based on user-provided predicates.

2 Overview

We illustrate our approach for verifying hyperproperties by reduction to CHC-SAT. We start with the simpler case of k-safety properties, followed by the more general case of $\forall^*\exists^*$ hyperproperties.

2.1 Motivating Example

As a means for highlighting the challenges in verifying hyperproperties, and, in particular, in reducing the problem to CHC solving, we present the example program `squaresSum` and its 2-safety specification from [41] in Fig. 1a. Given positive integers $a < b$, the program computes the sum of squares of all integers in the interval $[a, b)$. `squaresSum` is monotone in the sense that as the input interval increases, so does the output c. Formally, this is a 2-safety property that requires that whenever two traces satisfy the pre-condition $[a_2, b_2) \subset [a_1, b_1)$, they also satisfy the post-condition $c_1 > c_2$, where variable indices correspond to the traces that they represent. This is a special case of k-safety, where the relational property is checked at the end of the executions. More generally, we consider k-safety properties where the relational property is specified at designated *observation points* (explained in Sec. 3).

To verify the 2-safety property, a prominent approach is to reduce the problem to a regular safety verification problem by composing the program with itself (known as "self composition"). There are (infinitely) many possibilities for aligning the traces in the composed system, and the alignment chosen has direct impact on the complexity of the inductive invariant needed to establish safety. For example, if the two traces of `squaresSum` are aligned in lockstep, then initially $c_1 = c_2$, after one step, $c_1 < c_2$, and only later on, $c_1 > c_2$. Showing that $c_1 > c_2$ at the end requires tracking the difference $c_1 - c_2$, which is a complex value because it involves the sum of squares itself. This cannot be captured by an inductive invariant in first-order logic using theories currently supported by automated solvers (*e.g.*, linear arithmetic) and is therefore beyond reach for state-of-the-art solvers. On the other hand, if the second trace, whose input is the smaller interval, "waits" for a_1 and a_2 to coincide before proceeding in lockstep, then the property that $c_1 > c_2$ becomes inductive (except for the first step), greatly simplifying the inductive invariant. It is therefore important to consider the alignment and the (relational) inductive invariant together.

The requirements that the alignment and inductive invariant need to satisfy can be formulated in first-order logic [41]. To do so, we denote the program variables by $V = \langle a, b, c\rangle$. We express the initial states and program steps as formulas over V (and primed variant V') : $\mathit{Init}(V) \triangleq a > 0 \land b > a \land c = 0$, $\mathit{Tr}(V, V') \triangleq a < b \land c' = c + a \cdot a \land a' = a + 1 \land b' = b$. To reason about two traces, we use two copies of V, denoted V_1 and V_2. We introduce "unknown" predicates $\mathit{Inv}, A_{\{1\}}, A_{\{2\}}, A_{\{1,2\}}$ over $\langle V_1, V_2\rangle$ to capture the inductive invariant and desired alignment of the traces. $\{A_u\}_u$ define an *arbiter* that, when A_u is satisfied, schedules the steps of the traces according to u (for example, schedule $u = \{1\}$ stands for a step in trace 1 and a stutter in trace 2). The arbiter therefore determines the alignment of the traces. The inductive invariant Inv relates states of the two copies of the program, making it *relational.*

The problem of searching for the alignment and the inductive invariant simultaneously is then posed as a satisfiability problem (modulo the theory of arithmetic) of the formulas in Fig. 1b. To ensure that the arbiter, which determines the alignment, does not avoid violations of the post-condition by making one of the traces stutter forever s.t. it never reaches its final state, formulas 5-7 require that the arbiter only schedule a trace if it has not exited the loop, unless both traces exited the loop (in which case both are scheduled). This "validity" requirement means that, at the latest, the arbiter must schedule a trace when the other reaches the final state. Formulas 1-4 then ensure that all states that are reachable, subject to the steps permitted by the arbiter, must satisfy Inv. Specifically, the first formula ensures the initiation condition of the inductive invariant: the invariant satisfies the pre-condition and includes all the initial states of the composed system. Formulas 2-4 ensure the consecution of the invariant under every choice the arbiter makes. The 8th formula ensures the safety of the invariant and the last formula mandates that there is always at least one choice that is enabled, and that the system never reaches a "stuck" state.

An interpretation for the unknown predicates $Inv, A_{\{1\}}, A_{\{2\}}, A_{\{1,2\}}$ defines an arbiter and a corresponding inductive invariant. A possible solution is

$$A_{\{1\}}(V_1, V_2) \triangleq a_1 < a_2 \lor (b_2 \leq a_1 < b_1) \qquad A_{\{2\}}(V_1, V_2) \triangleq \bot$$
$$A_{\{1,2\}}(V_1, V_2) \triangleq (a_1 = a_2 \land a_1 < b_2) \lor a_1 \geq b_1$$
$$Inv(V_1, V_2) \triangleq 0 < a_1 \leq b_1 \land 0 < a_2 \leq b_2 \land \big((a_1 < a_2 \land c_1 \geq c_2) \lor (a_1 \geq a_2 \land c_1 > c_2)\big)$$

This solution captures the arbiter that makes the second trace wait until $a_1 = a_2$, then makes both traces proceed together until the second one exits its loop, in which case the first trace continues to execute alone until it also exits its loop and both traces are again (vacuously) scheduled together. The solution to Inv captures the corresponding inductive invariant previously discussed.

2.2 Challenges in Encoding Hyperproperty Verification as CHC-SAT

The formulas of Fig. 1b, with the exception of the last one, are constrained *Horn* clauses. That is, when the implications in these formulas are converted to disjunctions, at most one predicate application appears positively in each clause.

Alas, the presence of the last formula precludes direct application of existing CHC solvers. The problem is the disjunction on the right hand side of the implication. Such a disjunction appears to be crucial for a correct encoding of the problem. The reason is that uninterpreted predicates designate semantic *relations*. With such predicates denoting the choice of schedule, it is easy to drop into a vacuous solution where some states have no corresponding choice and are essentially "stuck", unsoundly making a post-condition violation unreachable. Encoding the requirement that every state have a schedule results in a clause with multiple occurrences of positive literals, capturing inherent disjunctions over the possible choices, which are not Horn. In particular, these disjunctions cannot be eliminated by renaming [37].

Previous works tackled this obstacle either by employing explicit enumeration of alignments that satisfy the non-Horn clause to avoid the disjunction [41], or by developing specialized techniques that are able to handle such disjunctions [45].

2.3 Our Approach: Transformation to CHC

In this paper, we show that the problem of searching for an alignment together with a (relational) inductive invariant can be encoded using CHCs, allowing us to reduce the problem to CHC-SAT, *without* fixing the alignment *a priori.*

A key insight of our reduction to CHC-SAT is the use of "doomed" states as a way to avoid the problematic disjunction over all choices of schedules. We refer to a given state as "doomed" if it necessarily reaches a state that violates

$$
\begin{aligned}
&D_{\{1\}}(V_1,V_2) \land D_{\{2\}}(V_1,V_2) \land D_{\{1,2\}}(V_1,V_2) \land \mathit{Init}(V_1) \land \mathit{Init}(V_2) \land a_2 > a_1 \land b_2 < b_1 \rightarrow \bot\\
&\neg(a_1 \geq b_1 \land a_2 \geq b_2 \rightarrow c_1 > c_2) \rightarrow D_{\{1\}}(V_1, V_2)\\
&\neg(a_1 \geq b_1 \land a_2 \geq b_2 \rightarrow c_1 > c_2) \rightarrow D_{\{2\}}(V_1, V_2)\\
&\neg(a_1 \geq b_1 \land a_2 \geq b_2 \rightarrow c_1 > c_2) \rightarrow D_{\{1,2\}}(V_1, V_2)\\
&\neg(a_1 < b_1) \rightarrow D_{\{1\}}(V_1, V_2)\\
&\neg(a_2 < b_2) \rightarrow D_{\{2\}}(V_1, V_2)\\
&\neg(a_1 < b_1 \land a_2 < b_2) \land \neg(a_1 \geq b_1 \land a_2 \geq b_2) \rightarrow D_{\{1,2\}}(V_1, V_2)\\
&D_{\{1\}}(V_1', V_2') \land D_{\{2\}}(V_1', V_2') \land D_{\{1,2\}}(V_1', V_2') \land \mathit{Tr}(V_1, V_1') \land V_2 = V_2' \rightarrow D_{\{1\}}(V_1, V_2)\\
&D_{\{1\}}(V_1', V_2') \land D_{\{2\}}(V_1', V_2') \land D_{\{1,2\}}(V_1', V_2') \land V_1 = V_1' \land \mathit{Tr}(V_2, V_2') \rightarrow D_{\{2\}}(V_1, V_2)\\
&D_{\{1\}}(V_1', V_2') \land D_{\{2\}}(V_1', V_2') \land D_{\{1,2\}}(V_1', V_2') \land \mathit{Tr}(V_1, V_1') \land \mathit{Tr}(V_2, V_2') \rightarrow D_{\{1,2\}}(V_1, V_2)
\end{aligned}
$$

Fig. 2: CHC encoding of Fig. 1a.

the hyperproperty along *every* valid alignment (as opposed to *some* in the direct encoding). Importantly, due to this conjunctive nature, doomed states lend themselves to a Horn encoding. If an initial state is identified as doomed (i.e., the CHCs are unsatisfiable), then the property is violated and a counterexample can be retrieved. Otherwise, if the set of initial states does not intersect the set of doomed states, then the hyperproperty is proved. Moreover, given an interpretation of the unknown predicates in which the initial states are not doomed, an alignment and a corresponding inductive invariant can be retrieved.

Based on this insight, in Sec. 4, we develop a general transformation of formulas of a certain form, to an equi-satisfiable set of CHCs. Furthermore, we provide a transformation of solutions between the two formulations (in both directions). The first-order formulas to which the transformation is applicable follow the overall structure of the formulas in Fig. 1b, but are somewhat more general. For example, some of the unknown predicates may have additional arguments, which turn out to be useful when considering a broader class of hyperproperties beyond k-safety ($\forall^*\exists^*$).

In Sec. 5 we apply the transformation of Sec. 4 to reduce k-safety verification to CHC-SAT. When applying the transformation on the formulas encoding our running example (Fig. 1b), we obtain the set of CHCs depicted in Fig. 2 over unknown predicates $D_{\{1\}}, D_{\{2\}}, D_{\{1,2\}}$.

In the CHCs of Fig. 2, an unknown predicate D_u represents states that are "doomed" if schedule u is chosen. The first CHC requires that no initial state that satisfies the pre-condition is completely doomed, i.e., for every such state there is a schedule for which it is not doomed. The remaining CHCs encode the properties of doomed states for each schedule. For example, the CHCs where $D_{\{1\}}$ is in the head (right hand side of the implication) imply that a state is doomed for schedule $\{1\}$ if: (a) it violates the post-condition, (b) it already exited the loop and hence trace 1 cannot be the only trace to be scheduled, or (c) it is the pre-state of a transition taken by 1 leading to a post-state that is doomed for every choice u.

A solution to the CHCs in Fig. 2 can be obtained from the solution to the formulas in Fig. 1b by $D_u \triangleq \neg(\mathit{Inv} \land A_u)$ for every $u \in \{\{1\}, \{2\}, \{1,2\}\}$.

More generally, in Sec. 4, we show a bi-directional transformation of solutions.

2.4 Beyond k-Safety

Our transformation to CHCs is not limited to an encoding of k-safety, but also generalizes to hyperproperties that use $\forall^*\exists^*$ quantification over traces, as presented in Sec. 6.

Hyperproperties with existential trace quantification become meaningful in the presence of nondeterminism in the program. For an example of such a property, consider a nondeterministic variant of `squaresSum` where the assignment `c += a * a` is replaced by `if (*) c += a * a`. That is, the increment of `c` may nondeterministically be skipped. We may now wish to verify that, if $[a_2, b_2) = [a_1, b_1)$, then for every trace from input $[a_1, b_1)$ there exists a trace from input $[a_2, b_2)$ such that when both terminate, $c_1 \neq c_2$. This is a $\forall\exists$-hyperproperty.

To verify such properties, a "witness" function is needed to map the universally quantified traces to the corresponding existentially quantified traces such that the body of the formula holds for the combination of the traces. Even if a witness function is known, to verify that the combination of the traces satisfies the body of the formula, we still need to find a proper alignment of the traces and an inductive invariant. As in the case of k-safety, these components are all interdependent, making it desirable to search for all of them together.

In general, the witness function for the existentially quantified traces may need to depend on the full universally quantified traces. However, [8] defines a sound but incomplete *game semantics*, in which the witness function essentially constructs the existentially quantified traces step-by-step, in response to moves of a "falsifier" who reveals the universally quantified traces step-by-step.

We show in Sec. 6.1 that the problem of searching for a step-by-step witness function, an alignment and a (relational) inductive invariant can be encoded in first-order logic, and the encoding is amenable to our transformation to CHCs. This results in a sound and complete CHC encoding of the game semantics of [8] for transition systems whose branching degree is bounded by a constant, which we henceforth refer to as "finite branching".

The idea in the $\forall^*\exists^*$-first-order encoding is to let the unknown predicates A_u specify not only the schedules chosen by the arbiter but also the choice of existentially quantified traces for the witness function. To do so, we assign a unique label to each of the possible transitions, and use these labels to identify the transitions along the traces. In this encoding, instead of u denoting a schedule only, it now denotes both a schedule and a choice of labels identifying the next transitions in the existentially quantified traces according to the witness function. Furthermore, the A_u predicates receive additional arguments that represent the next labels along the universally quantified traces.

For example, in the nondeterministic variant of `squaresSum`, there are at most two possible transitions in each control location. We therefore introduce two labels to distinguish between these possibilities: `i` for "increment" and `s` for "skip". The predicates that describe the schedules and the choices of existentially quantified traces for the $\forall\exists$-hyperproperty of interest are:

$$A_{\{1\},\mathtt{i}}, A_{\{2\},\mathtt{i}}, A_{\{1,2\},\mathtt{i}}, A_{\{1\},\mathtt{s}}, A_{\{2\},\mathtt{s}}, A_{\{1,2\},\mathtt{s}}.$$

They are defined over $\langle V_1, V_2, a\rangle$, where a ranges over the possible labels.

Note that in this encoding, the A_u predicates are no longer defined over $\langle V_1, V_2\rangle$ only, but have additional arguments for the labels of the universally quantified traces, while *Inv* does not. Thus, the reduction to CHCs applies our transformation in a more general setting than Fig. 1b. Furthermore, since u denotes both a schedule and a choice of labels for the existentially quantified traces, the number of A_u predicates depends on the number of labels. To ensure that there are finitely many predicates, we require the transition system to have a finite branching degree (otherwise, the space of possible labels becomes infinite).

Finally, in Sec. 6.2, we extend our approach to handle infinite (or unbounded) branching in the transition system, which can result, for example, from reading an input from an infinite domain. To do so, we introduce another first-order encoding that roughly replaces the infinitely-many concrete choices of transitions by finitely-many abstract choices. Unlike the cases of k-safety and $\forall^*\exists^*$-hyperproperties with finite branching, the resulting encoding is sound but incomplete w.r.t. the game semantics. By applying our transformation, we obtain a sound (albeit incomplete) reduction to CHC-SAT.

3 Background

We use first-order logic to model systems and their properties. Throughout the paper, we fix a background first-order theory $\mathcal{T}$ and denote its signature by Σ.

Transition Systems A (symbolic, labeled) *transition system* is a tuple $TS = (V, a, Init, Tr)$, where V is a vocabulary, *i.e.*, a vector of (logical) variables, each associated with a sort from Σ, denoting state variables; a is a label variable; *Init* is a formula over Σ with free variables V, and *Tr* is a formula over Σ with free variables $V \cup \{a\} \cup V'$, where V' consists of the primed variants of V.

A state of *TS* is a valuation to V, and we denote by $\mathbb{S}$ the set of all such valuations; $\mathbb{L}$ is the set of values that a can take, called labels; $\mathbb{S}_0 \subseteq \mathbb{S}$ is the set of initial states, which consists of all valuations that satisfy *Init*, and $\mathbb{R} \subseteq \mathbb{S} \times \mathbb{L} \times \mathbb{S}$ is the transition relation, which consists of the valuations for the composite vocabulary $V \cup \{a\} \cup V'$ that satisfy *Tr*. For simplicity, we assume that $\mathbb{R}$ is total, *i.e.*, $\forall s \in \mathbb{S}\ \exists \ell \in \mathbb{L}, s' \in \mathbb{S} \cdot (s, \ell, s') \in \mathbb{R}$[3]. We say that *TS* is *deterministic* when $\forall s \in \mathbb{S}, \ell \in \mathbb{L} \cdot \big|\{s' \mid \mathbb{R}(s, a, s')\}\big| = 1$ and that it has finite branching when $\mathbb{L}$ is finite. A trace of *TS* is an infinite sequence of states $t = s_0, s_1, \ldots$ such that for every $i \geq 0$ there exists $\ell \in \mathbb{L}$ such that $(s_i, \ell, s_{i+1}) \in \mathbb{R}$. We denote by $t[i]$ the i'th state in t. We further denote the set of traces that start from a state s by $\mathbb{T}(s)$, and the set of all traces of *TS* by $\mathbb{T}$.

Hyperproperties and their specification. We consider a fragment of the relational logic OHyperLTL [6] , which we call $\forall^*\exists^*$-OHyperLTL with formulas of the form:

$$\varphi \;=\; \psi \rightarrow \forall \pi_1 : \xi_1, \ldots, \pi_l : \xi_l \cdot \exists \pi_{l+1} : \xi_{l+1}, \ldots, \pi_k : \xi_k \cdot \; \Box\phi$$

[3] w.l.g.; *Tr* can always be replaced by $Tr \vee ((\forall a\, \forall V' \cdot \neg Tr) \wedge V' = V)$, which corresponds to adding self loops to states that have no outgoing transition.

where π_i are trace variables whose intended valuations are taken from $\mathbb{T}$; ξ_i are (non-temporal) formulas with free variables V that determine *observation points* along the k traces, where the traces must synchronize; ψ is a pre-condition that is assumed to hold initially; and ϕ needs to globally hold when all traces reach the observation points (which they must synchronize on before moving on). V_j denotes a copy of V where all variables are indexed by j. We refer to the variables in V_j as the state variables of the j'th trace (namely, π_j). When $l = k$, *i.e.*, all quantifiers are universal, φ is a *k-safety* property. A relational pre/post specification, as used in our motivating example, is a special case of a k-safety property where the observable points are the final states (which are augmented with self loops). For example, Fig. 1a presents the $\forall^*\exists^*$-OHyperLTL specification of the motivating example. When $l < k$, the formula also includes existential quantifiers, extending expressiveness to include some hyperliveness properties. An example of a security hyperliveness property that can be expressed in $\forall^*\exists^*$-OHyperLTL is *generalized non-interference (GNI)* [38]. GNI requires that for any two traces π_1 and π_2 there exists a trace π_3 whose high (secret) inputs agree with π_1 and whose low (public) inputs and outputs agree with π_2.

$\forall^*\exists^*$-OHyperLTL formulas are interpreted over transition systems. Intuitively, φ holds in a transition system if from every k initial states that jointly satisfy the pre-condition ψ, for every l traces from the first l states there exist corresponding $k-l$ traces from the remaining $k-l$ states s.t. the composed states of all traces globally satisfy ϕ, when the traces are projected to their observation points. Formally, given a transition system *TS* and φ as above, we refer to a tuple $(s_1, \ldots, s_k)$ of k states of *TS* as a *composed state*. A composed state defines a valuation to $V_1 \cup \ldots \cup V_k$, where s_j is the valuation of V_j. A composed state is initial if $s_i \in \mathbb{S}_0$ for every $1 \leq i \leq k$. We say that $TS \models \varphi$ if for every initial composed state $\overline{s} = (s_1, \ldots, s_k)$ such that $\overline{s} \models \psi$ the following holds: for every $t_1, \ldots, t_l \in \mathbb{T}(s_1) \times \cdots \times \mathbb{T}(s_l)$ there exist $t_{l+1}, \ldots, t_k \in \mathbb{T}(s_{l+1}) \times \cdots \times \mathbb{T}(s_k)$ such that $(\!|t_1|\!)_{\xi_1}, \ldots, (\!|t_k|\!)_{\xi_k} \models \Box\phi$, where $(\!|t_i|\!)_{\xi_i}$ is the projection (filtering) of trace t_i to states satisfying ξ_i. The semantics of $\Box\phi$ is that $t'_1, \ldots, t'_k \models \Box\phi$ iff $\forall i \leq \min(|t'_1|, \ldots, |t'_k|) \cdot (t'_1[i], \ldots, t'_k[i]) \models \phi$. Note that the semantics is oblivious to the transition labels since labels are only implicit in traces. Labels become useful in Sec. 6, where we use them to identify transitions along traces.

Remark 1. To simplify the presentation we consider hyperproperties defined w.r.t. a single transition system. The extension to multiple transition systems is straightforward. Similarly, $\Box\phi$ can be generalized to any temporal safety property via the standard automata-theoretic approach to model checking.

Constrained Horn Clauses (CHCs) are defined over a signature Σ' that extends Σ with a set $\mathcal{P}$ of (uninterpreted) predicates. Symbols in Σ are called *interpreted*, while the predicates in $\mathcal{P}$ are *uninterpreted* (sometimes called *unknown*). First-order formulas over Σ are called *constraints*. A CHC is a first-order formula of the form $\forall \mathcal{X} \cdot \bigwedge_i P_i(\mathcal{X}_i) \land \varphi(\mathcal{X}) \rightarrow H(\mathcal{X}_H)$ where $\mathcal{X}$ is a vector of (logical) variables; $P_i \in \mathcal{P}$ (not necessarily distinct, *i.e.*, it is possible that $P_{i_1} = P_{i_2}$ for

$$\begin{aligned}
&\alpha(\mathcal{V}) \to \mathit{Inv}(\mathcal{V})\\
&\mathit{Inv}(\mathcal{V}) \wedge \beta(\mathcal{V}) \to \bot\\
\boxed{\forall}\quad &\mathit{Inv}(\mathcal{V}) \wedge A_u(\mathcal{V},\mathcal{W}) \wedge \gamma_u(\mathcal{V},\mathcal{W}) \to \bot\\
\boxed{\forall}\quad &\mathit{Inv}(\mathcal{V}) \wedge A_u(\mathcal{V},\mathcal{W}) \wedge \delta_u(\mathcal{V},\mathcal{V}',\mathcal{W}) \to \mathit{Inv}(\mathcal{V}')\\
&\mathit{Inv}(\mathcal{V}) \to \bigvee_{u\in U} A_u(\mathcal{V},\mathcal{W})
\end{aligned}$$

$(\boxed{\forall} = \forall u \in U)$ (a)

$$\begin{aligned}
&\bigwedge_{u\in U} D_u(\mathcal{V},\mathcal{W}) \wedge \alpha(\mathcal{V}) \to \bot\\
\boxed{\forall}\quad &\beta(\mathcal{V}) \to D_u(\mathcal{V},\mathcal{W})\\
\boxed{\forall}\quad &\gamma_u(\mathcal{V},\mathcal{W}) \to D_u(\mathcal{V},\mathcal{W})\\
\boxed{\forall}\quad &\bigwedge_{u'\in U} D_{u'}(\mathcal{V}',\mathcal{W}') \wedge \delta_u(\mathcal{V},\mathcal{V}',\mathcal{W}) \to D_u(\mathcal{V},\mathcal{W})
\end{aligned}$$

(b)

Fig. 3: Formula scheme before (a) and after (b) the transformation.

$i_1 \neq i_2$); H is either $\bot$ or a predicate from $\mathcal{P}$; $\mathcal{X}_i, \mathcal{X}_H \subseteq \mathcal{X}$; and φ is a constraint. The universal quantification over $\mathcal{X}$ is often omitted.

A set of CHCs (or, more generally, first-order formulas) is *satisfiable* (modulo $\mathcal{T}$) if it has satisfying model M such that the projection of M onto Σ is a model of $\mathcal{T}$. A *solution* to a set of CHCs maps every predicate in $\mathcal{P}$ to a formula over Σ that defines it such that substituting all occurrences of the predicates by their definitions results in formulas that are valid modulo $\mathcal{T}$. If a set of CHCs has a solution then it is satisfiable. However, the converse may not hold due to the limited expressive power of first-order formulas.

4 General Transformation to CHCs

In this section we describe a satsifiability-preserving transformation that lets us convert a set of formulas, which adheres to a specific FOL scheme, to an equi-satisfiable set of CHCs. An extended version, with step-by-step details of the transformation, appears in [34]. Later we show how verification of a $\forall^*\exists^*$-OHyperLTL property can be captured by a set of formulas of the aforementioned scheme, where this transformation allows us to then reason about the correctness of the $\forall^*\exists^*$-OHyperLTL property by deciding the satisfiability of the CHCs.

Consider the scheme in Fig. 3a for a set of formulas over a signature Σ' that extends the signature Σ of the background theory by unknown predicates *Inv* and $\{A_u\}_{u\in U}$, for some finite set U. $\mathcal{V}, \mathcal{V}', \mathcal{W}$ denote disjoint vocabularies, *i.e.*, vectors of (logical) variables that are implicitly universally quantified. A row prefixed by $\boxed{\forall}$ indicates $|U|$ formulas, where u is substituted by all corresponding values from U. $\alpha, \beta, \gamma_u, \delta_u$ designate *constraints* (no occurrence of *Inv* or A_u).

At a high level, formulas 1 and 4 in Fig. 3a use *Inv* to capture an inductive invariant of the "states" (valuations to $\mathcal{V}$) reachable from α by "transitions" of δ_u, restricted according to a choice $u \in U$ of an "arbiter" $\{A_u\}_u$. Formula 2 establishes the fact that the reachable states are disjoint from some "bad states" β. Formulas 3 allow to enforce that the arbiter meets certain requirements, and formula 5 ensures that the arbiter makes a choice for every "state" in *Inv*.

Example 1. For our running example, we have $\mathcal{V} = \langle V_1, V_2\rangle = \langle a_1, b_1, c_1, a_2, b_2, c_2\rangle$, $\mathcal{V}' = \langle V_1', V_2'\rangle = \langle a_1', b_1', c_1', a_2', b_2', c_2'\rangle$, and $\mathcal{W} = \langle\rangle$ (The extra vocabulary $\mathcal{W}$ will come into use later in the paper). U is the set of arbitration choices $\{\{1\},\{2\},\{1,2\}\}$, and the corresponding completion of the constraint holes $\alpha, \beta, \gamma_u, \delta_u$ is easily discernible. (Note that a constraint on the right of $\rightarrow$ corresponds to its negation on the left.)

Note that the last formula in Fig. 3a is not a CHC since its head is a disjunction of unknown predicates. To remedy this shortcoming, we transform the formulas in Fig. 3a into the set of CHCs in Fig. 3b. The CHCs obtained for the running example are included in the extended version of the paper [34]. The transformation ensures:

Theorem 1. *The set of formulas in Fig. 3a is equi-satisfiable to the system of CHCs in Fig. 3b. Furthermore, there is an efficient translation of models of the former to models of the latter, and vice versa.*

Proof. The extended version of the paper [34] includes a stepwise transformation that shows how the CHCs in Fig. 3b are obtained from the formulas in Fig. 3a, where each step preserves equi-satisfiability and models. Here, due to space constraints, we only describe the final translation between models, which we have verified with Z3:

Given $Inv, A_u \models$ [Fig. 3a]	Given $D_u \models$ [Fig. 3b]
$D_u(\mathcal{V},\mathcal{W}) \triangleq \neg(Inv(\mathcal{V}) \wedge A_u(\mathcal{V},\mathcal{W}))$	$Inv(\mathcal{V}) \triangleq \forall \mathcal{W} \cdot \bigvee_{u\in U} \neg D_u(\mathcal{V},\mathcal{W})$
	$A_u(\mathcal{V},\mathcal{W}) \triangleq \neg D_u(\mathcal{V},\mathcal{W})$

5 Encoding *k*-Safety Verification as CHCs

In this section we address the problem of verifying k-safety properties via a CHC encoding. To this end, we start with a natural, non-Horn, encoding of the problem, as described in the previous section and previous works [41,45,8], and apply our transformation to obtain an equi-satisfiable system of CHCs.

Consider the k-safty formula: $\varphi \;=\; \psi \rightarrow \forall \pi_1 : \xi_1, \ldots, \pi_k : \xi_k \cdot \;\Box\phi$

This formula holds in a transition system TS if, starting from initial composed states that satisfy the pre-condition ψ, the observable states along every tuple of k traces satisfy ϕ, when the observable states are reached synchronously. Note that a pre-post specification, as used in our motivating example, is a special case of such a formula where the observable states are the final states. Verifying φ corresponds to finding (1) an alignment of the traces that synchronizes the observation points defined by $\xi_1, \ldots, \xi_k$, and (2) an inductive invariant that establishes that ϕ holds whenever $\xi_1, \ldots, \xi_k$ hold. Note that the invariant needs to be inductive along the aligned traces, including intermediate states between observable points. As different alignments give rise to different inductive invariants, it is desirable to find both of them simultaneously [41].

As before, we model the alignment using an arbiter that schedules a subset $\varnothing \neq M \subseteq \{1, \ldots, k\}$ of the traces to make a step based on the current composed state $s_1 \cdots s_k$. The arbiter may be nondeterministic, but it must choose at least

$$\begin{array}{lr}
 & \bigwedge_i \mathit{Init}(V_i) \land \psi(\mathcal{V}) \to \mathit{Inv}(\mathcal{V}) \\
 & \mathit{Inv}(\mathcal{V}) \land \mathit{Bad}(\mathcal{V}) \to \bot \\
\boxed{\forall} & \mathit{Inv}(\mathcal{V}) \land A_M(\mathcal{V}) \land \neg \mathit{valid}_M(\mathcal{V}) \to \bot \\
\boxed{\forall} & \mathit{Inv}(\mathcal{V}) \land A_M(\mathcal{V}) \land \delta_M(\mathcal{V},\mathcal{V}') \to \mathit{Inv}(\mathcal{V}') \\
 & \mathit{Inv}(\mathcal{V}) \to \bigvee_M A_M(\mathcal{V}) \\
(\boxed{\forall} = \forall M) & \\
 & \text{(a)}
\end{array}$$

$$\begin{array}{lr}
 & \bigwedge_M D_M(\mathcal{V}) \land \bigwedge_i \mathit{Init}(V_i) \land \psi(\mathcal{V}) \to \bot \\
\boxed{\forall} & \mathit{Bad}(\mathcal{V}) \to D_M(\mathcal{V}) \\
\boxed{\forall} & \neg \mathit{valid}_M(\mathcal{V}) \to D_M(\mathcal{V}) \\
\boxed{\forall} & \bigwedge_{M'} D_{M'}(\mathcal{V}') \land \delta_M(\mathcal{V},\mathcal{V}') \to D_M(\mathcal{V}) \\
 & \text{(b)}
\end{array}$$

Fig. 4: k-safety formula scheme before (a) and after (b) the transformation.

one set M. Furthermore, the arbiter must respect the synchronization of the observation points: it must not let a trace proceed beyond its observation point before the other traces reached theirs. This motivates the following definition.

Definition 1 (valid schedules). *M is a* valid schedule *for a composed state $s_1 \cdots s_k$ if either of the following two conditions holds:*
1. $\forall i \in M \cdot s_i \not\models \xi_i$ *2.* $\forall i \in M \cdot s_i \models \xi_i$ *and* $M = \{1, ..., k\}$.

Intuitively, the observation points act as a "barrier". All traces must reach the observation point before any of them can progress past it; and when they do, they do it simultaneously.[4]

To reason about composed states, we define a vocabulary $\mathcal{V} = V_1 \cup \ldots \cup V_k$ that consists of the set of state variables of all traces. We encode the arbiter using a family of unknown predicates $\{A_M(\mathcal{V})\}_M$ for every $\varnothing \neq M \subseteq \{1, \ldots, k\}$ and the inductive invariant using an unknown predicate $\mathit{Inv}(\mathcal{V})$. We express the situation where all traces reach an observable state but ϕ does not hold using the constraint: $\mathit{Bad}(\mathcal{V}) \triangleq \bigwedge_i \xi_i(V_i) \land \neg\phi(\mathcal{V})$. The joint steps of the traces as determined by the schedule M are given by the following constraint:

$$\begin{aligned}
\varDelta_M(\mathcal{V}, \mathcal{V}', a_1, \ldots, a_k) &\triangleq \textstyle\bigwedge_{i \in M} \mathit{Tr}(V_i, a_i, V_i') \land \bigwedge_{i \notin M} V_i = V_i' \\
\delta_M(\mathcal{V}, \mathcal{V}') &\triangleq \exists a_1, \ldots, a_k \cdot \varDelta_M(\mathcal{V}, \mathcal{V}', a_1, \ldots, a_k)
\end{aligned}$$

Note that the label variables are existentially quantified[5], indicating that any labeled transition can be used. The definition of a valid schedule is captured by:

$$\mathit{valid}_M(\mathcal{V}) \triangleq \begin{cases} \bigwedge_{i \in M} \neg\xi_i(V_i) & M \neq \{1, \ldots, k\} \\ \left(\bigwedge_{i \in M} \neg\xi_i(V_i)\right) \lor \left(\bigwedge_{i \in M} \xi_i(V_i)\right) & M = \{1, \ldots, k\} \end{cases} \quad (1)$$

Fig. 4a formalizes the joint requirements of the arbiter and the inductive invariant that ensures that φ holds. The following theorem summarizes the soundness of the encoding, which is a slight generalization of the encoding in [41] (where only pre/post specifications are considered):

[4] The requirement that all traces leave the observation point in tandem saves the need to record which of them already made a step since the last observation point.

[5] Since δ_M appears on the left-hand side of an implication, existential quantifiers can be pushed outside as universal quantifiers, resulting in quantifier-free bodies.

```
1  sum = 0;
2  b = *;
3  if (b > 0) {
4    i = 0;
5    while (i < n - 1) {
6      sum = sum + A[i];
7      i++;
8    }
9  }
10 else {
11   i = 1;
12   while (i < n) {
13     y = *;
14     sum = sum + A[i] + y;
15     i++;
16   }
17 }
```

$(A_1 = A_2 \wedge n_1 = n_2) \rightarrow \forall \pi_1 : pc = 5 \; \exists \pi_2 : pc = 5 \vee pc = 12 \cdot \square (b_2 \leq 0 \wedge sum_1 = sum_2)$

Fig. 5: Example for a $\forall\exists$ hyperproperty.

Theorem 2. *The set of formulas in Fig. 4a is satisfiable iff* $TS \models \varphi$*.*

Example 2. Applying the scheme of Fig. 4a to the program and $\forall^*\exists^*$-OHyperLTL specification of the 2-safety property from Fig. 1a results in Fig. 1b, except for moving constraints to the right hand side of the implication when it assists readability. Note that in this example, the observation points ξ_i of both traces correspond to the condition for exiting the loop (which is the negated loop condition). As a result $valid_{\{i\}} \triangleq a_i < b_i$ for $i \in \{1, 2\}$ and $valid_{\{1,2\}} \triangleq (a_1 < b_1 \wedge a_2 < b_2) \vee (\neg(a_1 < b_1) \wedge \neg(a_2 < b_2))$.

The set of formulas in Fig. 4a fits the general scheme of Fig. 3a; Thus, it is amenable to our general satisfiability-preserving transformation, the CHCs in Fig. 4(b). Since the transformation is satisfiability preserving, we obtain the following as a corollary of Thm. 1 and 2:

Corollary 1. *The system of CHCs in* Fig. 4b *is satisfiable iff* $TS \models \varphi$*.*

Where $A_M(\mathcal{V})$ in Fig. 4a describes the states where choosing schedule M leads to successful verification with *Inv* as an inductive invariant, $D_M(\mathcal{V})$ in Fig. 4b can be understood as describing states where choosing M would *prevent* the verification from going through in the sense that no inductive invariant would exist. In other words, these states are "doomed" if M is chosen, hence the choice of notation. If the set of CHCs in Fig. 4b is satisfiable, it proves that initial states that satisfy the pre-condition are not doomed. This intuition can be interpreted in a dual manner: if the initial states are not doomed, then there exists an alignment for which a safe inductive invariant exist.

6 Encoding $\forall^*\exists^*$ Hyperproperties as CHCs

In this section we consider the more general case of $\forall^*\exists^*$-OHyperLTL specifications. Throughout the section, TS is a transition system, and we fix a formula:

$$\varphi \; = \; \psi \rightarrow \forall \pi_1 : \xi_1, \ldots, \pi_l : \xi_l \cdot \exists \pi_{l+1} : \xi_{l+1}, \ldots, \pi_k : \xi_k \cdot \; \square \phi$$

In order to encode the problem of deciding if $TS \models \varphi$ as a satisfiability problem, we follow [8], and consider a *game semantics*, which is natural due to the alternation of quantifiers. The $\forall$ and $\exists$ quantifiers are "demonic" and "angelic", thus controlled by the falsifier and the verifier, respecitvely.

In the following, we introduce the game semantics of [8] for $\forall^*\exists^*$-OHyperLTL. We then encode truth of φ in TS under the game semantics as a satisfiability problem, and use the transformation from Sec. 4 to obtain a system of CHCs that is satisfiable iff TS satisfies φ according to the game semantics.

Example 3. To illustrate the game semantics, we use the example in Fig. 5, which accompanies this section. The presented program computes the sum of an array slice, nondeterministically choosing between the slice $A[0..n-2]$ and $A[1..n-1]$. For the second variant, an arbitrary integer can be added to each summand. This allows the program to fulfill the specification at the bottom, which requires that for every execution there is a corresponding execution of the second variant (where $b_2 \leq 0$) such that the sums at lines 5 and 12 align at every iteration. The specification is valid because y at line 13 can always be chosen to compensate for the deviation due to the index i not being the same.

Considering the game semantics, the falsifier first has to choose a value for b, which can be either positive or nonpositive. If it is nonpositive, then the verifier wins the game vacuously because $\xi_1 \triangleq (pc = 5)$ is never reached. If the choice is positive, then the verifier must choose nonpositive to satisfy $b_2 \leq 0$ from the specification. In subsequent steps, the verifier must select a scheduling that will align $pc_1 = 5$ and $pc = 12$ at every iteration, and select a value for y such that after both assignments (lines 6 and 14) $sum_1 = sum_2$ is satisfied. When following these choices, the verifier manages to satisfy $sum_1 = sum_2$ at all observation points, which gives it a winning play.

Safety games are played between a *verifier*, whose goal is to avoid *bad* states, and a *falsifier* who tries to reach a bad state. Formally, the game is a tuple $\mathcal{G} = (VS, FS, S_0, \delta_V, \delta_F, B)$ where VS are *verifier states*, in which the verifier moves, and FS are *falsifier states*, in which the falsifier moves, and $VS \cap FS = \varnothing$. The *game states* are $S = VS \cup FS$. $S_0 \subseteq S$ is a set of initial states, and $B \subseteq S$ is a set of bad states. $\delta_V \subseteq VS \times S$ defines the possible moves of the verifier and $\delta_F \subseteq FS \times S$—of the falsifier. It is assumed that δ_V, δ_F are total *i.e.*, there is at least one move for each player from every state. A *play* is a sequence of game states $\sigma_0, \sigma_1, \ldots$ such that $\sigma_0 \in S_0$, and for every $i \geq 0$, $(\sigma_i, \sigma_{i+1}) \in \delta_V \cup \delta_F$. The play is winning for the verifier if it is infinite and $\sigma_i \notin B$ for every $i \geq 0$. A (memoryless) strategy for the verifier is a function $\chi : VS \to S$ such that $(\sigma, \chi(\sigma)) \in \delta_V$ for every $\sigma \in VS$. χ is a *winning strategy* for the verifier if all the plays in which the verifier moves according to χ are winning for the verifier.

Game semantics for $\forall^*\exists^*$*-OHyperLTL* Let φ be as above. The game that captures the semantics of φ is defined with respect to a deterministic labeled transition system $TS = (V, a, Init, Tr)$. (We can always determinize TS by extending the set of labels without affecting the semantics; this step may introduce infinitely many labels, which do not require any special treatment in the definition of the game, but whose CHC encoding will be addressed in Sec. 6.2.)

The game for φ and TS proceeds in rounds, where in each round the falsifier makes a move and the verifier responds. The falsifier states are composed states (of k traces), and the verifier states augment them with a record of the falsifier's last move. The bad states are falsifier states where all traces are in their observation points but ϕ does not hold. The falsifier is responsible for choosing the transitions that define the $\forall$ traces $t_{1..l}$ assigned to $\pi_{1..l}$. The verifier responds by choosing the transitions of the $\exists$ traces $t_{l+1..k}$ assigned to $\pi_{l+1..k}$. Here the labels of the transitions come into play: the players specify the transitions of choice by picking a label $\ell \in \mathbb{L}$ for each trace. (Since TS is deterministic, transitions are uniquely identified by labels.) The traces then need to be aligned s.t. they synchronize on their observation points defined by ξ_i. The alignment does not affect the winner of the play, as long as it is a valid alignment. However, as in the case of k-safety, the alignment is instrumental for obtaining a winning strategy that has a simple description. As a result, the choice of the (valid) alignment is also left to the verifier. Altogether, a move of the falsifier consists of picking labels $\ell_l, \ldots, \ell_l \in \mathbb{L}$ for the $\forall$ trace variables; a move of the verifier consists of picking a valid subset $\varnothing \neq M \subseteq \{1, ..., k\}$ of the traces to progress (as in Sec. 5) and also labels $\ell_{l+1}, \ldots, \ell_k \in \mathbb{L}$ for the $\exists$ trace variables, and proceeding to the resulting composed state.[6] In this manner, the verifier iteratively "reads off" the states of $t_{1..l}$, properly aligned, and generates the traces $t_{l+1..k}$, while avoiding the bad states. If the verifier can do so indefinitely, then this proves that φ holds.

Formally, the components of the game are as follows (here, M represents a valid schedule according to Definition 1):

$$FS = \mathbb{S}^k \qquad VS = \mathbb{S}^k \times \mathbb{L}^l \qquad S_0 = \{\overline{s} \in \mathbb{S}_0^k \mid \overline{s} \models \psi\}$$
$$B \;= \{\overline{s} \in FS \mid \overline{s} \not\models \phi \text{ and } s_i \models \xi_i \text{ for every } 1 \leq i \leq k\}$$
$$\delta_F \;= \{(\overline{s}, (\overline{s}, \overline{\ell}^{\forall})) \mid \overline{s} \in FS,\ \overline{\ell}^{\forall} \in \mathbb{L}^l\} \quad \delta_V = \{((\overline{s}, \overline{\ell}^{\forall}), \overline{s}') \mid \overline{s} \overset{M,\overline{\ell}}{\rightsquigarrow} \overline{s}' \text{ for } \overline{\ell}^{\exists} \in \mathbb{L}^{k-l}\}$$

The notation $\overline{s} \overset{M,\overline{\ell}}{\rightsquigarrow} \overline{s}'$ indicates that $\overline{s}'$ is obtained from $\overline{s}$ by taking the transition with label ℓ_i from s_i whenever $i \in M$, and stuttering otherwise, where $\overline{\ell} = \langle \ell_1, \ldots, \ell_k \rangle$. We refer to it as a transition of the composed system according to schedule M labeled $\overline{\ell}$. The labels are split into $\overline{\ell}^{\forall} = \langle \ell_1, .., \ell_l \rangle$ and $\overline{\ell}^{\exists} = \langle \ell_{l+1}, .., \ell_k \rangle$. Formally, $\quad \overline{s} \overset{M,\overline{\ell}}{\rightsquigarrow} \overline{s}' \iff \bigwedge_{i \in M} \mathbb{R}(s_i, \ell_i, s_i') \wedge \bigwedge_{i \notin M} s_i = s_i'$.

Example 4. In the example of Fig. 5, the labels of transitions are integer values that reflect the choice of `*` at lines 2 and 13 (and have no effect on other states). The verifier and falsifier specify their moves using these labels. For example, in order to ensure that $sum_1 = sum_2$ is satisfied at every iteration, the verifier selects a transition label $\ell = A[i-1] - A[i]$ in line 13, which sets the value of y accordingly; after both assignments at lines 6 and 14, $sum_1 = sum_2$ holds.

The game semantics of $\forall^*\exists^*$-OHyperLTL is based on the winner in the verification game:

Definition 2 (Game Semantics for $\forall^*\exists^*$-OHyperLTL [8]). *Let TS be a transition system and φ a $\forall^*\exists^*$-OHyperLTL formula. TS* satisfies φ according

[6] In [8], steps of the verifier are split to two. Our definition is more precise in the sense that a winning strategy in the game of [8] implies a winning strategy in our game.

to the game semantics, *denoted $TS \models_{\mathcal{G}} \varphi$, if the verifier has a winning strategy in the verification game $\mathcal{G}_{TS,\varphi}$.*

Theorem 3 (as shown in [8]**).** *If $TS \models_{\mathcal{G}} \varphi$ then $TS \models \varphi$.*

6.1 CHC Encoding of the Game with Finite Branching

To encode the verification game for φ and TS, we introduce unknown predicates $\{A_u\}_{u\in U}$ that describe the strategy of the verifier as well as an unknown predicate *Inv* that encodes an inductive invariant that ensures that the strategy is winning. We first consider the case where the set of labels $\mathbb{L}$ is finite, i.e., TS has a finite branching. This makes it possible to define U as the set of all possible concrete choices of the verifier and introduce a predicate A_u per every possible choice of the verifier. To do so, we define $U = \mathbb{M} \times \mathbb{L}^{k-l}$, where $\mathbb{M} = \mathcal{P}(\{1,\ldots,k\}) \setminus \{\varnothing\}$ is the set of possible schedules, and $\mathbb{L}^{k-l}$ are the choice labels for constructing the traces assigned to $\{\pi_i\}_{i=l+1..k}$. Note that U is finite in this case. For each $u = \langle M, \overline{\ell^{\exists}}\rangle \in U$, the predicate A_u describes the verifier states in which the verifier chooses u for its move. Recall that verifier states consist of both the previous state of the verifier, captured by the composed state vocabulary $\mathcal{V}$ defined as before, and the last move of the falsifier, captured by label variables $\langle a_1,\ldots,a_l\rangle$. We denote $\mathcal{L}^{\forall} = \langle a_1,\ldots,a_l\rangle$, $\mathcal{L}^{\exists} = \langle a_{l+1},\ldots,a_k\rangle$ and $\mathcal{L} = \mathcal{L}^{\forall} \cup \mathcal{L}^{\exists} = \langle a_1,\ldots,a_k\rangle$. Then, the A_u predicates are defined over $\mathcal{V} \cup \mathcal{L}^{\forall}$. The *Inv* predicate is defined over $\mathcal{V}$ only, as it describes a set of falsifier states.

The formulas in Fig. 6a formalize the requirements that ensure that $\{A_u\}_u$ defines a winning strategy for the verifier, while accounting for the alternating choices of the falsifier ($\overline{\ell^{\forall}}$) and verifier ($\langle M, \overline{\ell^{\exists}}\rangle$) in every round, where

$$\varDelta_M(\mathcal{V},\mathcal{V}',\mathcal{L}) \quad = \textstyle\bigwedge_{i\in M} Tr(V_i,a_i,V_i') \wedge \bigwedge_{i\notin M} V_i = V_i'$$

$$\delta_{M,\overline{\ell^{\exists}}}(\mathcal{V},\mathcal{V}',\mathcal{L}^{\forall}) = \varDelta_M(\mathcal{V},\mathcal{V}',\mathcal{L})\big[\mathcal{L}^{\exists} \mapsto \overline{\ell}'\big] \qquad Bad(\mathcal{V}) \;=\; \textstyle\bigwedge_i \xi_i(V_i) \wedge \neg\phi(\mathcal{V})$$

$\varDelta_M$ is the formula expression of $\overset{M,\overline{\ell}}{\rightsquigarrow}$ from above. That is, $\overline{s}, \overline{s}', \overline{\ell}$ (valuations to $\mathcal{V},\mathcal{V}',\mathcal{L}$) satisfy $\varDelta_M$ if the composed system according to M has a transition from $\overline{s}$ to $\overline{s}'$ labeled $\overline{\ell}$. $\delta_{M,\overline{\ell^{\exists}}}$ is then the projection of $\varDelta_M$ to a concrete choice of labels $\overline{\ell^{\exists}}$ for the existentially quantified traces; the labels for the universals, captured by $\mathcal{L}^{\forall}$, remain free.

Theorem 4. *The set of formulas in Fig. 6a is satisfiable iff $TS \models_{\mathcal{G}} \varphi$.*

Proof. A solution for Fig. 6a induces a winning strategy χ for the verifier in the game for φ and TS: $\chi(\overline{s}, \overline{\ell^{\forall}}) = \overline{s}'$ for $\overline{s} \models Inv$, where $\overline{s}'$ is reached by choosing $\langle M, \overline{\ell^{\exists}}\rangle$ (*i.e.*, $\overline{s}, \overline{s}', \overline{\ell^{\forall}} \models \delta_{M,\overline{\ell^{\exists}}}$) such that $\overline{s}, \overline{\ell^{\forall}} \models A_{M,\overline{\ell^{\exists}}}$; such $\overline{s}'$ must exist because the last formula states that there must always be a choice for the verifier in falsifier states that satisfy *Inv*. For $\overline{s} \not\models Inv$, $\chi(\overline{s}, \overline{\ell^{\forall}})$ is defined arbitrarily. In the other direction, given a winning strategy for the verifier, we define the interpretation of *Inv* to be its winning region and the interpretation of $A_{M,\overline{\ell^{\exists}}}$ to consist of the falsifier states $(\overline{s}, \overline{\ell^{\forall}})$ where the strategy chooses $\overline{s}'$ such that $\overline{s}, \overline{\ell^{\forall}} \models A_{M,\overline{\ell^{\exists}}}$.

$$\begin{aligned}
&\textstyle\bigwedge_i \mathit{Init}(V_i) \land \psi(\mathcal{V}) \to \mathit{Inv}(\mathcal{V})\\
&\mathit{Inv}(\mathcal{V}) \land \mathit{Bad}(\mathcal{V}) \to \bot\\
\boxed{\forall}\quad &\mathit{Inv}(\mathcal{V}) \land A_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{L}^\forall) \land \neg \mathit{valid}_M(\mathcal{V}) \to \bot\\
\boxed{\forall}\quad &\mathit{Inv}(\mathcal{V}) \land A_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{L}^\forall) \land \delta_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{V}',\mathcal{L}^\forall) \to \mathit{Inv}(\mathcal{V}')\\
&\mathit{Inv}(\mathcal{V}) \to \bigvee_{\langle M,\overline{\ell}^\exists\rangle \in U} A_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{L}^\forall)
\end{aligned}$$

(a)

$$\begin{aligned}
&\bigwedge_{\langle M,\overline{\ell}^\exists\rangle \in U} D_{M,\overline{\ell}^\exists}(\mathcal{V}) \land \bigwedge_i \mathit{Init}(V_i) \land \psi(\mathcal{V}) \to \bot\\
\boxed{\forall}\quad &\mathit{Bad}(\mathcal{V}) \to D_{M,\overline{\ell}^\exists}(\mathcal{V})\\
\boxed{\forall}\quad &\neg \mathit{valid}_M(\mathcal{V}) \to D_{M,\overline{\ell}^\exists}(\mathcal{V})\\
\boxed{\forall}\quad &\bigwedge_{\langle M',\overline{\ell}'^\exists\rangle \in U} D_{M',\overline{\ell}'^\exists}(\mathcal{V}') \land \delta_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{V}') \to D_{M,\overline{\ell}^\exists}(\mathcal{V})
\end{aligned}$$

(b)

Fig. 6: A game formula scheme before (a) and after (b) the transformation, where $\boxed{\forall} = \forall\langle M,\overline{\ell}^\exists\rangle \in U$.

Remark 2. For k-safety properties, the encoding in Fig. 6a, based on the game semantics, is equivalent to the encoding in Fig. 4a (Sec. 5). In particular, in this case, the set $\mathcal{L}^\exists$ is empty, which means that $\overline{\ell}^\exists = \langle\rangle$, resulting in a game with finite branching, namely only the choices of the schedule M. Note that for such properties, the benefits of the game semantics are less obvious since if $TS \models \varphi$, then *every* strategy is winning for the verifier.

Encoding safety games in general The game encoding in Fig. 6a and Thm. 4 are stated here for the specific safety games corresponding to $\forall^*\exists^*$-OHyperLTL verification in order to avoid additional notational burden. However, the result is applicable to a more general class of safety games where the moves of the players are organized in rounds, each of which comprises of a move of the falsifier, followed by a move of the verifier. Furthermore, the states of the verifier are "intermediate states" defined as $VS = FS \times \Omega$, where Ω is a set of *auxiliary states* used to record the last falsifier move. The initial and bad states are falsifier states. The verifier moves to a new state according to the previous state together with the auxiliary state, while the falsifier is only allowed to choose the auxiliary part of the state. Therefore, $\delta_F \subseteq \{\langle \hat{s}, \langle \hat{s}, \omega\rangle\rangle \mid \hat{s} \in FS, \omega \in \Omega\}$. The encoding extends to such games, where $\mathit{Init}(V_i) \land \psi(\mathcal{V})$ is replaced by an encoding of S_0; Bad is replaced by an encoding of B; $\delta_{M,\overline{\ell}^\exists}(\mathcal{V},\mathcal{V}',\mathcal{L}^\forall)$ is replaced by an encoding of $\delta_V \circ \delta_F$ as a formula where the falsifier state variables and the choices of the falsifier are free, and $\mathit{valid}_M(\mathcal{V})$ is replaced by a guard encoded over the same free variables that ensures that the verifier step is applicable. Accordingly, our subsequent results (including the CHC encoding) extend to any such game.

Applying our transformation to the formulas in Fig. 6a results in the CHCs in Fig. 6b. Intuitively, $A_{M,\overline{\ell}^{\exists}}$ describe the winning strategy for the verifier: for "safe" states, represented by *Inv*, and given a move made by the falsifier, if the verifier chooses to move according to $\langle M, \overline{\ell}^{\exists}\rangle$, then it stays in the "safe" region. In contrast, $D_{M,\overline{\ell}^{\exists}}$ represents "doomed" states. Namely, if the verifier chooses to move according to $\langle M, \overline{\ell}^{\exists}\rangle$ from a state in $D_{M,\overline{\ell}^{\exists}}$, then the falsifier can force reaching a bad state for every choice of the verifier in the next steps of the game.

Corollary 2. *The set of CHCs in Fig. 6b is satisfiable iff* $TS \models_{\mathcal{G}} \varphi$.

Example 5. The example in Fig. 5 fits the case of finite branching if we assume that the integer values in the array A and those of *sum* and t are bounded modulo 2^m, and so are the labels $\mathbb{L}$. This means that the falsifier has 2^m possible steps at each game state, and the verifier has $3 \cdot 2^m$ (3 is the number of possible schedules out of $\{1, 2\}$). In the next subsection we explain how to encode the problem when the integers are considered to be unbounded.

6.2 CHC Encoding of the Game with Infinite Branching

The set of formulas in Fig. 6a, and the corresponding system of CHCs in Fig. 6b is well defined when the set U is finite. However, if $\mathbb{L}$ is infinite, so is U. In this case, instead of using $\mathbb{L}^{k-l}$ to specify the traces chosen by the verifier, we define a finite, abstract set of composed labels, denoted $\mathbb{L}^{\sharp}$, to be used by the verifier (the falsifier will continue to use the concrete labels to specify his transitions of choice). Each abstract label in $\mathbb{L}^{\sharp}$ is a relational predicate p with free variables $\mathcal{V}$ (the composed vocabulary) that relates the states of different traces. Thus, the vector of individual existential choices $\overline{\ell}^{\exists}$ of the verifier is now replaced with a *single* choice of a (relational) predicate $p \in \mathbb{L}^{\sharp}$ over all the copies. In contrast to the use of concrete labels to specify the (unique) next transition for each trace individually, an abstract label $p \in \mathbb{L}^{\sharp}$ determines the next transitions for the $\exists$ traces by relating their post-states to the rest of the composed post-state.

Specifically, given a set of labels $\overline{\ell}^{\forall}$ for the $\forall$ traces and a schedule M, a predicate $p \in \mathbb{L}^{\sharp}$ is used as a *restriction* (inspired by the homonymous concept from [8]) of the transitions of the composed system according to schedule M with $\forall$-choices $\overline{\ell}$, restricting the set of aforementioned transitions to those where the composed post-state satisfies p.

Example 6. In Fig. 5, at line 13, a nondeterministic integer value is assigned to variable t. Since the set of integers is infinite, assigning a unique label ℓ to each integer results in an infinite set $\mathbb{L}$. To specify the choices of the verifier, we therefore define a finite set of abstract labels. An example of such a set is $\mathbb{L}^{\sharp} = \{sum_1 = sum_2, sum_1 = y_2, sum_1 < y_2, sum_1 = sum_2 + A_2[i_2] + y_2\}$.The restriction $sum_1 = sum_2$ can result in an empty set of transitions (we will return to this point later in the section); but the restrictions $sum_1 = y_2$, $sum_1 < y_2$ and $sum_1 = sum_2 + A_2[i_2] + y_2$ always define a nonempty set of transitions when $pc_2 = 13$ and when a schedule $\{2\} \subseteq M$ is chosen: those transitions that choose a value for y_2 such that the predicate holds after the transition; there is always at least one such value. In fact, for $sum_1 = y_2$ and $sum_1 = sum_2 + A_2[i_2] + y_2$ there is exactly one such value, while for $sum_1 < y_2$, the set of values (transitions) is infinite. Note that there are transitions that are not selected by any restriction (those that assign to y_2 a value such that none of the predicates hold).

Thus, the abstract labels define a space of *underapproximations* of the transitions of the composed system. This is an underapproximation since some (combinations of) individual transitions of *TS* may not be allowed by any $p \in \mathbb{L}^\sharp$.

The verifier uses $p \in \mathbb{L}^\sharp$ to specify the transitions of the traces assigned to the existentially quantified variables $\pi_{l+1..k}$. We then require that *all* of the composed post-states reached by the verifier's choice $\langle M, p\rangle$ are winning for the verifier. This amounts to proving that *all* restricted traces satisfy $\Box\phi$, which would mean that there *exist* traces that do, *as long as the restrictions do not lead to an empty set of traces.* Therefore, to ensure soundness of the encoding, we require that the restrictions be nonempty. Nonemptiness of the restrictions also ensures that the choices of the falsifier are never restricted, since the choices of the falsifier are always singletons (based on the concrete labels).

Rather than limiting the set of predicates used as abstract labels, we ensure nonemptiness by applying the restrictions only when the resulting set of transitions is nonempty; otherwise, the full set of transitions is considered. Technically, this is accounted for by special considerations in the construction of the CHC encoding, as detailed below.

CHC encoding We adapt the formulas in Fig. 6a to the case of abstract labels. We define $U = \mathbb{M} \times \mathbb{L}^\sharp$. The formulas from Fig. 6a carry over, except that the definition of $\delta_{M,\overline{\ell}}$ from the finite branching case is now replaced with $\delta_{M,p}$, which captures the transitions according to the abstract labels, as defined below.

For a schedule $\varnothing \neq M \subseteq \{1..k\}$ and $p \in \mathbb{L}^\sharp$, we define $allowed_{M,p}$, a formula that is satisfied by $\overline{s}, \overline{\ell}^\forall$ when *some transition* is possible from $\overline{s}$ with scheduling M and $\forall$-choice $\overline{\ell}^\forall$ such that the target composed state satisfies p. This means that the restriction to p is nonempty. $\delta_{M,p}$ then applies the restriction of the composed post-state to p only when allowed (otherwise all transitions remain):

$$allowed_{M,p}(\mathcal{V}, \mathcal{L}^\forall) \triangleq \exists \mathcal{V}', \mathcal{L}^\exists \cdot \Delta_M(\mathcal{V}, \mathcal{V}', \mathcal{L}) \land p(\mathcal{V}')$$
$$\delta_{M,p}(\mathcal{V}, \mathcal{V}', \mathcal{L}^\forall) \triangleq \big(\exists \mathcal{L}^\exists \cdot \Delta_M(\mathcal{V}, \mathcal{V}', \mathcal{L})\big) \land \big(allowed_{M,p}(\mathcal{V}, \mathcal{L}^\forall) \to p(\mathcal{V}')\big)$$

The resulting encoding is sound, but, unlike the case of finite branching, not complete.

Theorem 5. *If the set of formulas in Fig. 6a adapted to $\mathbb{L}^\sharp$ is satisfiable, then $TS \models_{\mathcal{G}} \varphi$.*

Example 7. Going back to the example in Fig. 5, choosing schedule $M = \{2\}$ and restriction $\ell^\sharp = (sum_1 = sum_2 + A_2[i_2] + y_2)$ when $pc_2 = 13$ ensures that the unique value of y_2 that satisfies the restriction is selected. With this value chosen, the assignment of the next line will produce a value of sum_2 that is equal to that of sum_1. This gives rise to the following winning strategy (at every iteration): (i) schedule $\{1\}$ with any restriction until $pc_1 = 7$; (ii) schedule $\{2\}$ until $pc_2 = 13$, then schedule $\{2\}$ again with $\ell^\sharp = (sum_1 = sum_2 + A_2[i_2] + y_2)$, then $\{2\}$ again with any restriction; (iii) conclude the iteration by scheduling $\{1, 2\}$. As explained, the inductive invariant $sum_1 = sum_2$ is preserved in this behavior, *and* there are no "stuck" states (since, by construction of $\delta_{M,p}$, empty restrictions are lifted to the full set of transitions).

As a corollary of Thm. 5, satisfiability of the aforementioned formulas ensures that $TS \models \varphi$. To obtain an equi-satisfiable CHC encoding, we apply the transformation of Sec. 4. The resulting CHC encoding consists of the formulas in Fig. 6b adapted to use $\mathbb{L}^\sharp$ in the same way the formulas in Fig. 6a are adapted.

Corollary 3. *If the set of CHCs in Fig. 6b adapted to $\mathbb{L}^\sharp$ is satisfiable, then $TS \models_{\mathcal{G}} \varphi$.*

7 Evaluation

We implemented our CHC-encoding approach in a tool called HyHorn, on top of Z3 [23] (4.12.0) through its Python API, using Spacer [35,31] as a CHC solver. HyHorn takes as input a CFG, or several CFGs, whose transitions are annotated with two-vocabulary first-order formulas, and constructs a formula expressing the transition relation Tr. The specification is provided as: (i) a quantifier prefix $\forall\forall$, $\forall\exists$, or $\forall\forall\exists$, (ii) observation points ξ_i and (iii) safety condition ϕ that must hold globally in all observations. From that, the CHC encoding (Sec. 5, Sec. 6) is constructed and passed to Spacer for solving. HyHorn supports all first-order theories supported by Spacer (in our experiments, we used the theories of integer arithmetic and arrays). HyHorn further provides the option to apply predicate abstraction with a user-provided set of predicates, same as [8]. The abstraction is incorporated into the CHC encoding using the implicit abstraction encoding [13]. Notably, many of the benchmarks shown here are solved by HyHorn *even without* an abstraction, that is, directly over the concrete state.

In the area of hyperproperty verification, there are already several tools present, and the objective of our evaluation is to compare with such. Still, the field is not mature enough to have a standardized specification format (as is the case with SMTLIB and SV-COMP, to name a few). As a result, each tool has its own, opinionated, format, which varies from logical formulas to control-flow graphs. This makes it technically difficult to compare results of multiple solutions. In particular, benchmarks taken from previous work come in a range of formats, dictated by the tools that introduced them. A few of the benchmarks were translated by previous authors and, thanks to their efforts, are available in more than one format. For the majority of them, manual work is required for translating the benchmarks, and, more importantly, there is no one accepted translation, and the translation can introduce artifacts in the evaluation.

This forced us to prioritize the comparisons in our experiments. We chose to focus on comparison with the most closely related tools to our work. These

k-safety	HyHorn		HyPA	PCSat	Pdsc
	PA	concrete			
double square NI	0.56	—	67.0	—	6.8
double square NI ff	0.12	—	5.3	1.5	/
half square NI	0.30	0.30	63.0	13.4	3.4
squares sum	0.17	3.41	70.4	360.7	2.8
squares sum (simplified)	0.10	0.30	17.2	/	/
array insert	0.86	13.4	/	/	18.5
array insert (simplified)	1.33	2.58	16.2	378.6	/
exp1x3	0.08	0.09	2.9	/	/
fig 3 [FV19]	0.03	—	7.9	/	/
fig 2 [BF22]	0.11	—	13.6	/	/
col item symm	0.49	0.49	14.9	/	/
counter det	0.46	—	6.2	/	/
mult equiv	0.29	—	14.2	/	/
mult equiv (simplified)	0.19	—	10.3	/	/
array int mod	0.13	—	/	/	58.2
mult dist [FV19]	2.25	—	/	/	/

$\forall^*\exists^*$	HyHorn		HyPA
	PA	concrete	
non-det add	1.45	2.80	3.3
counter sum	0.09	—	4.0
async GNI	0.36	0.37	3.8
compiler opt 1	0.14	0.19	1.8
compiler opt 2	0.17	0.78	2.0
refine	0.18	0.29	4.0
refine 2	0.28	0.65	3.9
smaller	0.16	0.96	2.0
counter diff	0.17	—	6.8
fig 3 [BF22]	0.81	—	9.9
P1 (simple)	0.19	0.59	1.4
P1 (GNI)	0.26	0.75	138.7
P2 (GNI)	8.50	6.65	12.8
P3 (GNI)	0.32	0.20	4.6
P4 (GNI)	0.77	0.63	27.7

Table 1: Experimental results for k-safety properties. Time is measured in seconds. "—" represents timeouts after 20 minutes. "/" denotes benchmarks not present in the respective tool's suite.
In benchmark names, [FV19] refers to [27]; [BF22] refers to [8].

are HyPA [8], Pdsc [41], and PCsat [45]. HyPA is the most recent tool, and has already collected benchmarks from various previous papers (including Weaver [27]); Pdsc and PCsat both use the same first-order encoding as our starting point and thus are also relevant. HyPA's benchmarks include, in particular, $\forall^*\exists^*$ examples such as GNI, and Pdsc targets non-trivial alignments, and, as such all of its benchmarks have non-lockstep alignments.

Benchmarks For the evaluation of our approach we use *the full sets* of benchmarks from HyPA [8] and Pdsc [41]. The benchmarks of HyPA are divided into k-safety benchmarks, which are adopted from [43,27,41,45], and $\forall^*\exists^*$ benchmarks, which include refinement properties for compiler optimizations, general refinement of nondeterministic programs and generalized non-interference (GNI). For two benchmarks, we include both a simplified version as given in [8], as well as the original example. The benchmarks of Pdsc include more non-lockstep examples, as well as all of the comparator benchmarks from [43]. The comparator examples consist of both safe and unsafe instances. Weaver [27] considers 12 additional (sequential) k-safety benchmarks. As an additional test case, we manually translated the running example from Weaver, which is a 3-safety property with a nontrivial alignment, and tested it with HyHorn – HyHorn solved it in 2.25 seconds when provided with a few simple predicates (inequalities between program variables). We believe that being the running example makes it a good representative of the remaining 12. This brings our benchmark suite to a total of 112 k-safety examples (16 in Table 1 plus 96 comparator benchmarks).

Experiments To demonstrate the effectiveness of HyHorn we compare to HyPA [8], the most recent approach of formal verification of $\forall^*\exists^*$-hyperproperties, which employs a construction using automata. To exhibit the benefits of the direct CHC encoding we also compare the k-safety examples to PCSat [45] and Pdsc [41]. Both encode the k-safety problem using FOL formulas as in Fig. 4a. PCSat uses a specialized solver for pfwCSP (a fragment of FOL that includes these formulas), while Pdsc solves the FOL formulas by enumerating alignments and using a CHC solver for each alignment. We do not compare to game solvers since, as reported by [8], state-of-the-art infinite-state game solvers, such as [26,2], which work without user-provided predicates, are unable to solve the benchmarks we consider.

We run HyHorn on the full set of benchmarks, and each of the other tools on the ones included in their benchmark suite. This is because each tool has its own input format: HyPA and Pdsc each has its own representation for the transition system and the property; PCSat accepts pfwCSP instances that are constructed manually. Some of the benchmarks are common to several tools.

All experiments are run on an AMD EPYC 74F3 with 32GB of memory. HyPA and PCSat are executed in Docker using their published artifacts[7].

Results The performance measurements of the tools for the k-safety benchmarks and for the $\forall^*\exists^*$ benchmarks are shown in Table 1. The results for the comparator examples are deferred to the extended version of the paper [34]. HyHorn is tested in two modes: with predicate abstraction ("PA") and without ("concrete"). HyPA and Pdsc require predefined predicates (the same predicates are used in all tools), while PCSat does not, but uses hints to solve 'array insert' and 'squares sum'. HyHorn solves almost all of the benchmarks with PA in under a second, outperforming previous approaches by up to two orders of magnitude; and also solves most of the benchmarks quickly without PA, esp. the $\forall^*\exists^*$ properties. In particular, HyHorn solves the two array benchmarks, while HyPA and PCSat do not support arrays and only solve simplified versions with integers. The runtime of HyHorn (both with and without predicates) on the comparator examples is similar to the runtime of Pdsc (see [34]), where HyHorn solves some benchmarks that Pdsc does not. (The other tools do not include these benchmarks.) On the unsafe examples, HyHorn provides a concrete counterexample, while Pdsc is only able to determine that there is no inductive invariant and alignment expressible with the given set of predicates.

8 Related Work

There is a large body of work studying verification of hyperproperties. While earlier verification techniques mostly focus on k-safety properties, or specific examples such as program equivalence, monotonocity, determinism [5,44,3,30,43,47] [24,27,41,1], lately verification of non-safety hyperproperties has been studied [4,16,45,7,8]. Below we discuss the works closest to ours.

[7] We evaluated HyHorn in Docker as well. There were no meaningful differences in runtime.

k-Safety Automatic verification of k-safety properties can be achieved by reducing the problem to a standard safety verification problem by means of self-composition [5], product-programs [3], and their derivatives [47,24]. Recently, however, it was identified that the alignment of the different copies has a substantial effect over the complexity of the verification problem [41,27,12]. Our approach is most related to the technique of Shemer *et al.* [41], which uses a semantic alignment that chooses which copy of the system performs a move based on the composed state of the different copies. They suggest an algorithm that iterates through the set of possible semantic alignments, such that in each iteration a CHC solver tries to prove the property, with the chosen alignmnet, using predicate abstraction. Unlike [41], HyHorn delegates the search for the alignment to the CHC solver, together with the search for the invariant, making the algorithm less dependent on predicate abstraction. Moreover, while [41] is restricted to k-safety only, our technique can handle k-safety as well as the more general $\forall^*\exists^*$-OHyperLTL.

$\forall^*\exists^*$ *Hyperproprties* Recently, verification of $\forall^*\exists^*$ hyperproperties has been studied, targeting both finite and infinite systems [45,16,8]. Unno *et al.* [45] present an approach based on an encoding of hyperproperties verification as satisfiability of formulas in FOL that extend Horn form with disjunctions, existential quantification and well founded relations. Deciding satisfiability of the generated set of formulas is based on a variant of the CEGIS framework. HyHorn is different as it encodes $\forall^*\exists^*$-OHyperLTL verification as a set of CHCs, which does not require a specialized solver and can use any off-the-shelf CHC solver. Coenen et.al. [16] suggested a game-based approach for verification of $\forall^*\exists^*$ properties over finite-state systems, which was then extended by Beutner *et al.* [8] to handle infinite-state systems. Similarly to [8], we use game semantics to solve $\forall^*\exists^*$ problems, but do not require building the game-graph in order to solve the game, instead reducing the game solution to satisfiability of CHCs. It is important to note that in the case of infinite branching degree, while the approach in [8] explicitly checks for emptiness of restrictions in hindsight, i.e., after they are used in a strategy, and removes them iteratively if needed, HyHorn embeds the emptiness requirements into the set of CHCs. Recently, [7] extended the game-based approach to use prophecy variables as a way to achieve completeness of the reduction to games. Extending our approach to this case is a promising avenue for future research.

Relational CHCs [40] present a method for discovering relational solutions to CHCs. Their setting is different: the inputs are CHCs that serve as the definition of the transitions, and synchronization is between sets of unknown predicates; at the current state, only lock-step semantics is considered. Furthermore, their algorithm extends and modifies Spacer [35], while our approach can use any CHC solver without modification.

Infinite-State Game Solving Our approach for verifying $\forall^*\exists^*$ hyperproperties is based on the game semantics of $\forall^*\exists^*$-OHyperLTL proposed in [16,8]. However, we do not propose a general game solving algorithm. Instead, we use the game semantics to come up with a first-order encoding of hyperproperty verification problems, which is then reduced to CHC solving. This allows us to use any CHC solver when solving the hyperproperty game. There is a large body of work on solving infinite-state games [21,9,46,26]. The game solving approach in [46] uses three-valued predicate abstraction to reduce the problem to finite-state game solving and requires to iteratively refine the controllable predecessor operator when computing candidate winning states. The approach in [26] targets games defined over the theory of linear real arithmetic and is based on an unrolling of the game and the use of Craig interpolants [18] to synthesize a winning strategy. The game solver in [2] is not restricted to a given FOL theory, but requires an interpolation procedure in order to compute sub-goals that are used to inductively split a game into sub-games. As reported by [8], game solving approaches [26,2], which work without a provided set of predicates, are unable to handle the infinite-state games for the benchmarks we consider. Moreover, the approaches in [26,16,2,8] cannot handle games that are defined using formulas over the theory of arrays, which are part of our benchmark. The approach of [9] to solving games over infinite graphs is based on reduction of games (including safety games) to CHCs. However, unlike the reduction presented in this paper, in [9] the games are encoded in a different fragment of Horn, namely ∀∃-Horn where the head predicates can contain existential quantifiers. More recently (and concurrently with our work), [25] proposed a new reduction of game solving to CHC solving. Their approach handles safety games in which the branching degree of the "safe" player (the verifier in our setting) is bounded. In contrast, our encoding supports also infinite branching with the restrictions mechanism. Moreover, they do not support predicate abstraction, which is crucial for solving some of our benchmarks.

Restrictions as Underapproximations The use of restrictions as underapproxmations of the transition relation, inspired by [8], corresponds to the use of must hyper-transitions [36] in abstract transition systems [42,19] and games [20,22]. Similarly to [29,17], we use such underapproximations to replace an existential quantifier by universal quantification *within* the restriction.

9 Conclusion

We introduced a translation of a family of non-Horn first-order formulas to CHCs. This translation led to the first CHC encoding of a simultaneous inference of an invariant and an alignment for verifying k-safety properties. While the transformation itself is rather simple, identifying it was not straightforward and alluded previous works on the topic. We have further extended the CHC encoding to infer a witness function for existentially quantified traces arising in the verification of $\forall^*\exists^*$-OHyperLTL properties. Our experiments exhibit significant improvement over state-of-the-art hyperproperty verifiers thanks to the existence of advanced off-the-shelf CHC solvers, whose efficacy is expected to improve even further. The approach shows promising capabilities in solving (many) hyeprproperty verification problems completely automatically. In some cases, predicates still have to be provided by the user, a limitation that we hope to overcome in the future by automatic inference of predicates. Applying (or extending) the transformation to obtain CHC encoding for other verification fragments is an interesting direction for future work.

Acknowledgment The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 2875/21 and No. 2117/23, and by the NSF-BSF grant No. 2018675.

References

1. Antonopoulos, T., Koskinen, E., Le, T. C., Nagasamudram, R., Naumann, D. A., and Ngo, M. An algebra of alignment for relational verification. *Proc. ACM Program. Lang. 7*, POPL (jan 2023).
2. Baier, C., Coenen, N., Finkbeiner, B., Funke, F., Jantsch, S., and Siber, J. Causality-based game solving. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I* (2021), A. Silva and K. R. M. Leino, Eds., vol. 12759 of *Lecture Notes in Computer Science*, Springer, pp. 894–917.
3. Barthe, G., Crespo, J. M., and Kunz, C. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings* (2011), pp. 200–214.
4. Barthe, G., Crespo, J. M., and Kunz, C. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings* (2013), S. N. Artëmov and A. Nerode, Eds., vol. 7734 of *Lecture Notes in Computer Science*, Springer, pp. 29–43.
5. Barthe, G., D'Argenio, P. R., and Rezk, T. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA* (2004), pp. 100–114.
6. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., and Sánchez, C. A temporal logic for asynchronous hyperproperties. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I* (2021), A. Silva and K. R. M. Leino, Eds., vol. 12759 of *Lecture Notes in Computer Science*, Springer, pp. 694–717.
7. Beutner, R., and Finkbeiner, B. Prophecy variables for hyperproperty verification. In *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022* (2022), IEEE, pp. 471–485.
8. Beutner, R., and Finkbeiner, B. Software verification of hyperproperties beyond k-safety. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I* (2022), S. Shoham and Y. Vizel, Eds., vol. 13371 of *Lecture Notes in Computer Science*, Springer, pp. 341–362.

9. Beyene, T. A., Chaudhuri, S., Popeea, C., and Rybalchenko, A. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014* (2014), S. Jagannathan and P. Sewell, Eds., ACM, pp. 221–234.
10. Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday* (2015), pp. 24–51.
11. Bjørner, N. S., McMillan, K. L., and Rybalchenko, A. On solving universally quantified horn clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings* (2013), F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, Springer, pp. 105–125.
12. Churchill, B. R., Padon, O., Sharma, R., and Aiken, A. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019* (2019), K. S. McKinley and K. Fisher, Eds., ACM, pp. 1027–1040.
13. Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings* (2014), E. Ábrahám and K. Havelund, Eds., vol. 8413 of *Lecture Notes in Computer Science*, Springer, pp. 46–61.
14. Clarkson, M. R., Finkbeiner, B., Koleini, M., Micinski, K. K., Rabe, M. N., and Sánchez, C. Temporal logics for hyperproperties. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (2014), M. Abadi and S. Kremer, Eds., vol. 8414 of *Lecture Notes in Computer Science*, Springer, pp. 265–284.
15. Clarkson, M. R., and Schneider, F. B. Hyperproperties. *J. Comput. Secur. 18*, 6 (2010), 1157–1210.
16. Coenen, N., Finkbeiner, B., Sánchez, C., and Tentrup, L. Verifying hyperliveness. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (2019), I. Dillig and S. Tasiran, Eds., vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 121–139.
17. Cook, B., and Koskinen, E. Reasoning about nondeterminism in programs. *SIGPLAN Not. 48*, 6 (jun 2013), 219–230.
18. Craig, W. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. of Symbolic Logic 22*, 3 (1957), 269–285.
19. Dams, D., and Namjoshi, K. S. The existence of finite abstractions for branching time model checking. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings* (2004), IEEE Computer Society, pp. 335–344.
20. de Alfaro, L., Godefroid, P., and Jagadeesan, R. Three-valued abstractions of games: Uncertainty, but with precision. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings* (2004), IEEE Computer Society, pp. 170–179.

21. de Alfaro, L., Henzinger, T. A., and Majumdar, R. Symbolic algorithms for infinite-state games. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings* (2001), K. G. Larsen and M. Nielsen, Eds., vol. 2154 of *Lecture Notes in Computer Science*, Springer, pp. 536–550.
22. de Alfaro, L., and Roy, P. Solving games via three-valued abstraction refinement. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings* (2007), L. Caires and V. T. Vasconcelos, Eds., vol. 4703 of *Lecture Notes in Computer Science*, Springer, pp. 74–89.
23. de Moura, L. M., and Bjørner, N. Z3: an efficient SMT solver. In *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* (2008), pp. 337–340.
24. Eilers, M., Müller, P., and Hitz, S. Modular product programs. In *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* (2018), pp. 502–529.
25. Faella, M., and Parlato, G. Reachability games modulo theories with a bounded safety player. *Proceedings of the AAAI Conference on Artificial Intelligence 37*, 5 (June 2023), 6330–6337.
26. Farzan, A., and Kincaid, Z. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang. 2*, POPL (2018), 61:1–61:30.
27. Farzan, A., and Vandikas, A. Automated hypersafety verification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (2019), I. Dillig and S. Tasiran, Eds., vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 200–218.
28. Fedyukovich, G., Kaufman, S. J., and Bodík, R. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017* (2017), D. Stewart and G. Weissenbacher, Eds., IEEE, pp. 100–107.
29. Godefroid, P., Nori, A. V., Rajamani, S. K., and Tetali, S. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010* (2010), M. V. Hermenegildo and J. Palsberg, Eds., ACM, pp. 43–56.
30. Godlin, B., and Strichman, O. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab. 23*, 3 (2013), 241–258.
31. Gurfinkel, A. Program verification with constrained horn clauses (invited paper). In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I* (2022), S. Shoham and Y. Vizel, Eds., vol. 13371 of *Lecture Notes in Computer Science*, Springer, pp. 19–29.
32. Hoder, K., and Bjørner, N. S. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings* (2012), A. Cimatti and R. Sebastiani, Eds., vol. 7317 of *Lecture Notes in Computer Science*, Springer, pp. 157–171.

33. Hojjat, H., and Rümmer, P. The ELDARICA horn solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018* (2018), N. S. Bjørner and A. Gurfinkel, Eds., IEEE, pp. 1–7.
34. Itzhaky, S., Shoham, S., and Vizel, Y. Hyperproperty verification as chc satisfiability. Available at `https://doi.org/10.48550/arXiv.2304.12588`.
35. Komuravelli, A., Gurfinkel, A., and Chaki, S. SMT-based model checking for recursive programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), pp. 17–34.
36. Larsen, K. G., and Liu, X. Equation solving using modal transition systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990* (1990), IEEE Computer Society, pp. 108–117.
37. Lewis, H. R. Renaming a set of clauses as a horn set. *J. ACM 25*, 1 (1978), 134–135.
38. McCullough, D. Noninterference and the composability of security properties. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988* (1988), IEEE Computer Society, pp. 177–186.
39. McMillan, K. L. Lazy annotation revisited. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 243–259.
40. Mordvinov, D., and Fedyukovich, G. Property directed inference of relational invariants. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019* (2019), C. W. Barrett and J. Yang, Eds., IEEE, pp. 152–160.
41. Shemer, R., Gurfinkel, A., Shoham, S., and Vizel, Y. Property directed self composition. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (2019), I. Dillig and S. Tasiran, Eds., vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 161–179.
42. Shoham, S., and Grumberg, O. Monotonic abstraction-refinement for CTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 546–560.
43. Sousa, M., and Dillig, I. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016* (2016), pp. 57–69.
44. Terauchi, T., and Aiken, A. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings* (2005), pp. 352–367.
45. Unno, H., Terauchi, T., and Koskinen, E. Constraint-based relational verification. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I* (2021), A. Silva and K. R. M. Leino, Eds., vol. 12759 of *Lecture Notes in Computer Science*, Springer, pp. 742–766.

46. Walker, A., and Ryzhyk, L. Predicate abstraction for reactive synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014* (2014), IEEE, pp. 219–226.
47. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., and Malik, S. Lazy self-composition for security verification. In *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* (2018), H. Chockler and G. Weissenbacher, Eds., vol. 10982 of *Lecture Notes in Computer Science*, Springer, pp. 136–156.
48. Zaks, A., and Pnueli, A. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings* (2008), pp. 35–51.
49. Zhu, H., Magill, S., and Jagannathan, S. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018* (2018), J. S. Foster and D. Grossman, Eds., ACM, pp. 707–721.

Program Analysis

Maximal Quantified Precondition Synthesis for Linear Array Loops

Sumanth Prabhu S[1,2](✉), Grigory Fedyukovich[3](✉), and Deepak D'Souza[2](✉)

[1] Tata Consultancy Services Research, Pune, India
[2] Indian Institute of Science, Bengaluru, India
[3] Florida State University, Tallahassee, USA
sumanth.prabhu@tcs.com, grigory@cs.fsu.edu, deepakd@iisc.ac.in

Abstract. Precondition inference is an important problem with many applications in verification and testing. Finding preconditions can be tricky as programs often have loops and arrays, which necessitates finding quantified inductive invariants. However, existing techniques have limitations in finding such invariants, especially when preconditions are missing. Further, maximal (or weakest) preconditions are often required to maximize the usefulness of preconditions. So the inferred inductive invariants have to be adequately weak. To address these challenges, we present an approach for maximal quantified precondition inference using an *infer-check-weaken* framework. Preconditions and inductive invariants are inferred by a novel technique called *range abduction*, and then checked for maximality and weakened if required. Range abduction attempts to propagate the given quantified postcondition backwards and then strengthen or weaken it as needed to establish inductiveness. Weakening is done in a syntax-guided fashion. Our evaluation performed on a set of public benchmarks demonstrates that the technique significantly outperforms existing techniques in finding maximal preconditions and inductive invariants.

1 Introduction

Many practical problems in software development, verification, and testing rely on good and nontrivial preconditions for programs. Preconditions can be considered as a constraint on a program's input or used to filter out input values of a program at run-time. While performing verification in a backward fashion, preconditions are used to summarize loops and functions. The *maximal* (or logically weakest) precondition is desirable in all these applications. Such preconditions can be derived in various methods [45,14,54,13,53,25,3,46].

However, precondition inference is known to be difficult for programs with unbounded loops, as it requires reasoning about possible behaviors in any any loop iteration. This necessitates the inference of *inductive invariants* that describe a set of states from which a new iteration can begin and cannot escape. This task becomes particularly challenging in the presence of data structures like arbitrarily-sized arrays. When reasoning about array elements, solvers are

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 245–274, 2024.
https://doi.org/10.1007/978-3-031-57267-8_10

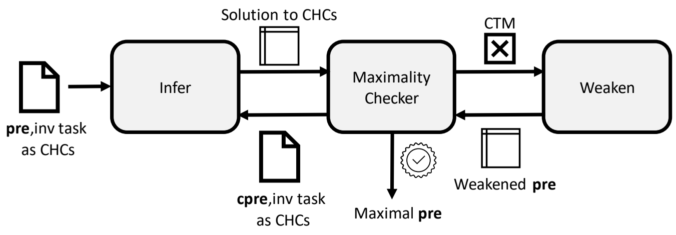


Fig. 1: An overview of our infer-check-weaken framework.

expected to support quantifiers, but existing techniques [30,40,34,32,22] have many limitations.

We present a new technique to automatically infer maximal quantified preconditions for deterministic programs that manipulate arrays and have *linear array loops*. These loops are non-nested and terminating loops with unique counter variables. The postconditions can have either universal or existential quantification. Since such programs can model many practical programs, several techniques target them, but for assertion checking and not precondition inference [39,10]. Moreover, we show in this paper that precondition inference is undecidable for this class of programs.

An overview of our algorithm is shown in Fig. 1. The algorithm operates in an "infer-check-weaken" framework. Our algorithm views the problem as solving a system of constrained Horn clauses (CHCs), which are logical systems to represent the verification conditions of programs, with a missing precondition $\boldsymbol{pre}$. A valid solution for this system is inferred by an abduction-based algorithm, i.e., by systematically answering the questions like "what state at the beginning of the iteration could yield a given state at the end of the iteration?" The solution is then checked for maximality by inferring another precondition ($\boldsymbol{cpre}$) for a system that uses the same CHC encoding of the loop and the complemented (i.e., negated) postcondition. If the solution is not maximal, it is weakened incrementally in a counter-example-guided loop.

The inference algorithm begins with the weakest possible candidate solution and propagates the given quantified postcondition towards the program's entry point. In the process, it strengthens the candidate solution using our novel technique called *range abduction*. Range abduction finds a strengthening of quantified formulas by reduction (wherever possible) to abduction over quantifier-free formulas. The obtained formulas are combined with the *range formula* [22] that essentially represents a boundary between the indices of arrays that are already processed and indices that are yet to be processed. Such a predicate can be obtained using lightweight static analysis over the structure of the CHCs. The inference algorithm uses the Houdini technique [24] to weaken a solution.

Intuitively, range abduction for linear array loops seeks to pose two integer abduction queries over indices that are modified and indices that are not modified. Integer abduction has been used in invariant inference [17,18], precondition inference [27,16], and specification synthesis [2,50]. On the lower level, abduction is often implemented using quantifier elimination, but in our setting the formulas must use quantifiers over array indices that should not be eliminated. Range abduction is designed specifically for this application.

Although efficient, range abduction does not guarantee maximality, and our inference is followed by two additional steps: *maximality checking* and *weakening*. The maximality checker tries to determine whether all the states outside the current precondition lead to a violation of the assertion. If they do, the current precondition is maximal. Otherwise, there is at least one state that can be added to the current precondition, and hence an attempt to weaken the precondition is made. The weakening module weakens a precondition and infers an inductive invariant for it using a syntax-guided-synthesis based method.

A prior framework [50] to find specifications (including preconditions) follows a similar approach by iteratively inferring solutions. But it is based on integer abduction and a maximality checking using an SMT solver, and it is applicable only to array-free programs. Furthermore, it does not guarantee maximality in some cases [29]. We experimentally observed that extending the SMT-based maximality checking algorithm of [50] with quantified formulas over arrays makes the tool diverge. This motivated us to design a new maximality checker by using ***cpre*** of the complement system and range abduction.

We have implemented our algorithm in a tool called PreQSyn, which takes CHCs as input. On a challenging set of 32 benchmarks, PreQSyn significantly outperforms a prior maximal quantified precondition inference tool P-Gen [54]. PreQSyn automatically found 31 preconditions and proved 21 of them to be maximal, while in contrast P-Gen found only 2 maximal preconditions and in most cases did not find any preconditions. We also show that a variety of existing array verification tools like VeriAbs [15], Spacer [32], and FreqHorn [22], find it hard to even *verify* the preconditions we discovered for these benchmarks. Our tool can not only solve them by finding preconditions, but also finds the maximal ones in most of the cases.

The core contributions of this paper are:

1. An algorithm, based on a new technique called range abduction, to infer universal and existential quantified preconditions and invariants, effective on linear array loop programs.
2. New methods to check maximality and weaken preconditions.
3. A tool that implements the algorithms to infer maximal preconditions and can be used as a CHC solver.
4. Experimental evaluation demonstrating the effectiveness of the algorithm.

In the rest of the paper, we motivate the problem with an example in Sect. 2. The necessary background for abduction and CHCs are provided in Sect. 3. A proof of undecidability of the problem is in Sect. 4. Sect. 5 presents an overview of our inference algorithm and an illustration on the example. The details of range abduction are in Sect. 6, while Sect. 7 has the maximality checking and weakening algorithms. Our experimental evaluation can be found in Sect. 8, related work in Sect. 9, and conclude with limitations and future work in Sect. 10.

```
int N = nondetInt();
int A[N], B[N], C[N];
assume(pre(A, B, C, N)); // goal: find maximal pre
for (int i = 0; i < N; i++)
  if (2*i < N) C[i] = i;
  else A[i] = C[i];
assert (∀j. 0 ≤ j < N ⟹ A[j] == B[j]); // postcondition
```

Fig. 2: C-like example with a universally quantified postcondition and no precondition.

$$\boldsymbol{pre}(N, A, B, C) \wedge i = 0 \Longrightarrow \boldsymbol{inv}(i,N,A,B,C) \qquad (C_1)$$
$$\boldsymbol{inv}(i,N,A,B,C) \wedge i < N \wedge 2 * i < N \wedge C' = store(C, i, i) \wedge i' = i+1 \Longrightarrow \boldsymbol{inv}(i',N,A,B,C') \qquad (C_2)$$
$$\boldsymbol{inv}(i,N,A,B,C) \wedge i < N \wedge 2 * i \geq N \wedge A' = store(A, i, C[i]) \wedge i' = i+1 \Longrightarrow \boldsymbol{inv}(i',N,A',B,C) \qquad (C_3)$$
$$\boldsymbol{inv}(i,N,A,B,C) \wedge \neg(i < N) \wedge \neg(\forall j.\, 0 \leq j < N \Longrightarrow A[j] = B[j]) \Longrightarrow \bot \qquad (C_4)$$

Fig. 3: CHC encoding of program in Fig. 2.

```
int N = nondetInt();
int A[N], B[N], C[N];
assume(cpre(A, B, C, N));
for (int i = 0; i < N; i++)
  if (2*i < N) C[i] = i;
  else A[i] = C[i];
assert (∃j. 0 ≤ j < N ∧ A[j] != B[j]); // complemented post
```

Fig. 4: The program used to check maximality; the postcondition is complemented and has no precondition.

2 Motivating Example

We motivate the problem with the program shown in Fig. 2 with three finite-length statically allocated arrays A, B, and C, each of the size N. The arrays are accessed sequentially in the loop: the cells in the first half of C are assigned their corresponding indices, and the remaining elements of C are copied to the corresponding positions in A. The program ends with the postcondition stating the pairwise equality of A and B. Our goal is to find the maximal precondition under which the postcondition holds. Intuitively, such a precondition must be universally quantified because it must express that arrays A, B, and C are properly initialized up to an arbitrary length N.

Further, in order to *prove* that the postcondition indeed holds after the loop has terminated, we have to show that there exists an *inductive invariant* that is also universally quantified. To confirm that the precondition is logically the weakest, we need to formally prove that any attempt to extend it by a single point leads to a violation of the postcondition. Thus, the solution we target should have two properties: 1) it should allow us to find an inductive invariant for the loop, and 2) any of its weakening results in a counterexample that violates the assertion.

The only publicly existing tool to find quantified precondition, P-Gen [54], which is based on predicate abstraction, is unable to solve this program. The last candidate precondition it tries to refine is $N = 3 \wedge A[0] = B[0] \wedge A[1] = B[1] \wedge A[2] = B[2]$, which does not constrain the value of array C, thus allowing the program to violate the postcondition, e.g., when $C[2] \neq B[2]$ initially and $A[2] = C[2]$ in the else-branch when $i = 2$.

Fig. 3 shows a system of CHCs over relations $\boldsymbol{pre}$ and $\boldsymbol{inv}$, representing the verification conditions of the program in Fig. 2. For brevity, we do not mention the universal quantification over all program variables including arrays, which is implicit. In particular, the first CHC identifies the initial value of the counter but does not give any constraints over A, B, or C (which are essentially deferred to $\boldsymbol{pre}$). The next two CHCs encode the loop body, corresponding to the two possible branches in the body of the loop. The last CHC encodes that no state satisfying the negation of the assertion is reachable.

The missing precondition makes the CHC system in Fig. 3 different from the CHC systems that appear in verification tasks. Hence, existing CHC solvers are not directly applicable here as they can return the strongest solution: $\perp$. For instance, Spacer [32] (Z3 v4.12.2) returns the solution $\boldsymbol{pre} \mapsto \lambda N, A, B, C.\perp$ and $\boldsymbol{inv} \mapsto \lambda i, N, A, B, C.\perp$. Such vacuous solutions are not of much use in the applications mentioned earlier.

The CHC system also represents a maximal specification problem, with $\boldsymbol{pre}$ being the specification of an initialization function. However, existing maximal precondition synthesis techniques [2,50,29] do not support synthesizing quantified preconditions over arrays.

Our algorithm takes the input CHC system and works in an *infer-check-weaken* fashion as shown in Fig 1. First, the infer module strengthens and weakens the postcondition from the last CHC via range abduction and Houdini, resp., to find the following precondition (detailed illustration follows in Sect. 5.2):

$$\lambda N, A, B, C.\, \forall j (0 \leq j < N \wedge 2 * j < N \implies A[j] = B[j]) \wedge \forall j (0 \leq j < N \wedge 2 * j \geq N \implies B[j] = C[j]).$$

We note that this is the maximal precondition for this problem instance, and in general we may not always find the maximal precondition in the first iteration. In any case, we need to check the maximality of the inferred precondition. Our maximality checker does this by trying to find a precondition for the complement of the postcondition (called the "complement program", see Fig 4). This is achieved by calling the infer module again, albeit with an existentially quantified postcondition. By using the existentially quantified structure of the postcondition, the infer module discovers the following precondition (see Sect. 6.4 for details):

$$\lambda N, A, B, C.\, \exists j (0 \leq j < N \wedge 2 * j \geq N \wedge B[j] \neq C[j]).$$

The maximality checker now tries to determine whether all the points that are outside the precondition of the original program are indeed in the precondition of the complement program. For the example program, this is encoded as the following formula:

$$\forall A, B, N.\ (\neg(\forall j(0 \leq j < N \wedge 2 * j < N \implies A[j] = B[j]) \wedge \forall j(0 \leq j < N \wedge 2 * j \geq N \implies B[j] = C[j])) \\ \implies \exists j(0 \leq j < N \wedge 2 * j \geq N \wedge B[j] \neq C[j])).$$

If the formula was valid, the current precondition would be maximal since all the states outside would violate the property (as they would be in the precondition of the complement program). In this example, the implication is not valid, because $N = 3$, $A = [0, 0, 0]$, $B = [1, 0, 0]$, $C = [0, 0, 0]$ is a counter-example to validity. Our approach then weakens the precondition of the complement program based on the counterexample to the validity check:

$$\lambda N, A, B, C.\ \exists j(0 \leq j < N \wedge 2 * j \geq N \implies B[j] \neq C[j]) \vee \exists j(0 \leq j < N \wedge 2 * j < N \wedge A[j] \neq B[j]).$$

The checker now conducts a successful validity check, and the algorithm terminates.

3 Background

This paper builds largely on foundations of *Satisfiability Modulo Theories* (SMT) problems. SMT aims to determine the existence of an assignment to variables of a first-order logic formula that makes it true. We will be dealing with the logical setting $\mathcal{L}$ of linear integer arithmetic (LIA) with arrays. The signature of the logic includes a finite set of uninterpreted relation symbols $\mathcal{R}$. Each symbol r in $\mathcal{R}$ has an associated arity a_r, and an associated type which indicates a type (integer or array) for each argument of the relation.

We write $\varphi(x_1, \ldots, x_n)$ (where each x_i is a variable with an associated integer/array type) to denote a formula φ of this logic, that does not use any of the relation symbols in $\mathcal{R}$, and whose free variables are among $\{x_1, \ldots, x_n\}$. For convenience, we also write $\varphi(\vec{x})$ to denote the same. For a formula $\varphi(\vec{x})$, and an assignment m which maps the variables in $\vec{x}$ to concrete integers/arrays, we write $m \models \varphi$ to denote that φ evaluates to $\top$ under m, and say m satisfies φ, or that m is a model of φ. A formula ψ is logically weaker than a formula φ (denoted $\varphi \implies \psi$), if every model of φ also satisfies ψ. Hence $\varphi \implies \bot$ denotes that φ is unsatisfiable.

An *interpretation* for a relation symbol $r \in \mathcal{R}$ is defined as a map of the form $\lambda x_1 \ldots \lambda x_n.\varphi(x_1, \ldots, x_n)$, where φ is a well-typed first-order formula that does not contain any symbols from $\mathcal{R}$.

We now present formal definitions of the concepts that will be used in the rest of the paper.

3.1 Abduction

Definition 1. *Let $\vec{x}$ and $\vec{y}$ be vectors of variables such that the variables in $\vec{x}$ are also present in $\vec{y}$. Let $\alpha(\vec{y})$ and $\beta(\vec{y})$ be formulas without any relation symbols from $\mathcal{R}$, with free variables in $\vec{y}$. Let r be an uninterpreted relation in $\mathcal{R}$ of arity equal to the length of $\vec{x}$. Consider a formula of the form $r(\vec{x}) \wedge \alpha(\vec{y}) \implies \beta(\vec{y})$. The abduction problem is to find an interpretation φ for r, such that:*

1. $\varphi(\vec{x}) \wedge \alpha(\vec{y}) \not\Longrightarrow \bot$, *and*
2. $\varphi(\vec{x}) \wedge \alpha(\vec{y}) \implies \beta(\vec{y})$.

Intuitively, the problem of abduction is to find a formula φ that together with α entails the formula β in a non-trivial manner. One can see that for a given abduction problem there may be multiple solutions, but we are interested in a maximal one (i.e. logically weakest), whenever a solution exists. The techniques in [17,16,2] compute such a maximal solution for first order theories that admit quantifier elimination. This solution is succinctly presented in the lemma below.

Lemma 1. *Let $r(\vec{x}) \wedge \alpha(\vec{y}) \implies \beta(\vec{y})$ be an abduction problem where the underlying first order theory has a method $\mathrm{QE}(\vec{q}, \psi)$ whose result is a $\vec{q}$-free formula constructed by (existential) quantifier elimination of variables $\vec{q}$ from the formula ψ. Suppose that the given instance has a solution. Then, the following formula $\varphi(\vec{x})$ forms a maximal solution for the abduction problem:*

$$\varphi(\vec{x}) \overset{\text{def}}{=} \neg(\mathrm{QE}(\vec{y} \setminus \vec{x}, \alpha \wedge \neg\beta)).$$

Example 1. Consider an instance of the abduction problem $r(x) \wedge y = 0 \implies x > y$. Then $\varphi(x)$ is computed as follows:

$$\varphi(x) = \neg(\mathrm{QE}(\{x, y\} \setminus \{x\}, y = 0 \wedge x \leq y)) =$$
$$\neg(\mathrm{QE}(y, y = 0 \wedge x \leq y)) = \neg(x \leq 0) = x > 0.$$

3.2 Modeling Programs With Constrained Horn Clauses

Constrained Horn clauses (CHCs) [37,28,57,35,51,23,31,50] are becoming increasingly popular as an intermediate logical representation of programs and their proof obligations. Dealing directly with CHCs as opposed to program statements is convenient and allows for easier creation and handling of various SMT formulas and constructed invariants.

Definition 2. *A CHC (in the logic $\mathcal{L}$) is a formula in $\mathcal{L}$ that has the form of one of the following three implications:*

$$\forall \vec{x_1}.\, (\varphi_1(\vec{x_1}) \implies r_1(\vec{x_1})) \tag{1}$$
$$\forall \vec{x}_1, \vec{x}_2.\, (r_1(\vec{x}_1) \wedge \varphi_2(\vec{x}_1, \vec{x}_2) \implies r_2(\vec{x}_2)) \tag{2}$$
$$\forall \vec{x}_1.\, (r_1(\vec{x}_1) \wedge \varphi_3(\vec{x}_1) \implies \bot) \tag{3}$$

where:

- $r_1, r_2 \in \mathcal{R}$ *are uninterpreted relation symbols, where* r_1 *and* r_2 *may coincide.*
- $\vec{x_1}, \vec{x_2}$ *are vectors of variables;*
- *the vectors* $\vec{x}_1$ *and* $\vec{x}_2$ *have no common elements, and*
- *the formulas* φ_i*, called* constraints, *have no uninterpreted symbols from* $\mathcal{R}$.

We introduce some auxiliary notation below for convenience. For a CHC C:

- $\mathit{body}(C)$ (resp. $\mathit{head}(C)$) denotes the left (resp. right) side of the implication in C,
- $\mathit{rel}(\mathit{body}(C))$ denotes the (singleton or empty) set of relation symbols in $\mathcal{R}$ that appear in $\mathit{body}(C)$,
- When C is of type (2), $\mathit{rel}(\mathit{head}(C))$ denotes the singleton set $\{r_2\}$ containing the relation appearing in $\mathit{head}(C)$,
- When C is of type (2), $\mathit{args}(\mathit{body}(C))$ (a.k.a. *source variables*) denotes $\vec{x}_1$ and $\mathit{args}(\mathit{head}(C))$ (a.k.a. *destination variables*) denotes $\vec{x}_2$.

A CHC of type (1) is called a *fact*, and of type (3) is called a *query.* For simplicity, for a query C, we write $\mathit{rel}(\mathit{head}(C)) = \mathit{rel}(\bot) = \bot$. In the literature, the CHCs we are considering are called *linear* as there is at most one relation symbol in the body of a CHC. A *system of CHCs* is a finite non-empty set of CHCs.

We assume that our precondition inference problem is represented by a system of CHCs S without any facts[4] and there is a designated relation $\boldsymbol{pre}$ (or $\boldsymbol{cpre}$) that appears in $\mathit{rel}(\mathit{body}(C))$ for some CHC C in S and doesn't appear in $\mathit{rel}(\mathit{head}(C'))$ for any other CHC C' in S. Furthermore, we assume that there is a single query in S with a constraint of the form $\varphi \wedge \rho$, where $\neg\rho$ is the postcondition in the inference problem.

CHCs allow for flexibility of program encoding. For instance, it is safe to assume that each φ_C is in Conjunctive Normal Form (CNF). For if C had the following form:

$$r_1(\vec{x}_1) \wedge (\varphi_1(\vec{x}_1, \vec{x}_2) \vee \varphi_2(\vec{x}_1, \vec{x}_2)) \implies r_2(\vec{x}_2),$$

it can be transformed into two CHCs:

$$\begin{aligned} r_1(\vec{x}_1) \wedge \varphi_1(\vec{x}_1, \vec{x}_2) &\implies r_2(\vec{x}_2) \\ r_1(\vec{x}_1) \wedge \varphi_2(\vec{x}_1, \vec{x}_2) &\implies r_2(\vec{x}_2). \end{aligned}$$

Definition 3 (CHC Solution and Satisfiability). *A* solution *to a system of CHCs S is a map $\mathcal{M}$ that provides an interpretation for each relation symbol in $\mathcal{R}$, such that for each CHC C in S, $\big(\mathit{body}(C) \implies \mathit{head}(C)\big)[\mathcal{M}/\mathcal{R}]$[5] is valid. In this case we say $\mathcal{M}$ is inductive at C. We say S is* satisfiable *if there exists a solution to it.*

Definition 4 (Maximal Precondition). *Let S be a system of CHCs for a precondition inference problem. We call a solution $\mathcal{M}$ to S* (precondition) maximal *if there is no solution $\mathcal{M}'$ to S with $\mathcal{M}'(\boldsymbol{pre})$ strictly logically weaker (i.e. w.r.t. the implication partial order) than $\mathcal{M}(\boldsymbol{pre})$. $\mathcal{M}(\boldsymbol{pre})$ is also called the weakest precondition.*

[4] A fact CHC represents the initial condition of the program. Since $\boldsymbol{pre}$ is in the place of initial condition in our task, there will not be a fact CHC.

[5] For a formula φ, terms/formulas a and b, we write $\varphi[b/a]$ to denote φ after all instances of a are replaced by b. For a set of terms/formulas X and a mapping $\mathcal{M}$ from X to other terms/formulas, $\varphi[\mathcal{M}/X]$ denotes the simultaneous replacement of all $x_1, x_2, \ldots \in X$ by $\mathcal{M}(x_1), \mathcal{M}(x_2), \ldots$, respectively.

We now define certain terms that will be used in weakening of a precondition (Sect. 7).

Definition 5 (Complement System). *Given a system of CHCs S, we define a complement system $\overline{S}$ to be the system obtained from S by replacing ρ by $\neg\rho$ in the query CHC.*

Definition 6 (CHC Extension). *Given a system of CHCs S with* **pre** *and an interpretation φ for* **pre***, we define S_φ, an extension of S w.r.t φ, to be the system obtained from S by replacing* **pre** *by φ in the CHC $C \in S$, where $rel(body(C)) = \{\textbf{pre}\}$*

Lemma 2. *Given S with* **pre** *and its extension S_φ, if $\mathcal{M}_\varphi$ is a solution to S_φ then $\mathcal{M} = \{\lambda r \in \mathcal{R}.\ \mathit{if}\ r = \textbf{pre}\ \mathit{then}\ \varphi\ \mathit{else}\ \mathcal{M}_\varphi(r)\}$ is a solution to S.*

To encode program executions, we borrow the notion of CHC unrolling from [21]. Essentially, a CHC unrolling is a symbolic representation of a set of program executions starting from a state satisfying φ. If the unrolling is satisfiable then the execution terminates in the postcondition.

Definition 7 (Unrolling of CHCs). *Given an extended CHC system S_φ over $\mathcal{R}$, let $C_0, \ldots, C_k$ be a $k+1$-length sequence of CHCs in S_φ, with C_0 being a fact, C_k being a query, and $rel(head(C_i)) = rel(body(C_{i+1}))$ for each i. Then, a k-length unrolling of S_φ is defined as below:*

$$\pi_{\langle C_0,\ldots,C_k\rangle} \overset{\text{def}}{=} \bigwedge_{0 \le i < k} body(C_i)(\vec{x}_i, \vec{x_{i+1}}) \wedge (body(C_k)[\neg\rho/\rho])(\vec{x}_k)$$

.

Example 2. Consider the CHC system S from Fig. 3. Let φ be:

$$(\forall j.\, 0 \le j < N \implies A[j] = B[j] = 0 \wedge C[j] = 1) \wedge N = 1$$

Then $\pi_{\langle C_1,C_2,C_4\rangle}$, which is a 3-length unrolling of S_φ, is the following satisfiable formula:

$$\begin{aligned}\pi_{\langle C_1,C_2,C_4\rangle} = {} & (\forall j.\, 0 \le j < N \implies A[j] = B[j] = 0 \wedge C[j] = 1) \wedge N = 1 \wedge i = 0 \wedge \\ & i < N \wedge 2 * i < N \wedge C' = store(C, i, i) \wedge i' = i + 1 \wedge \\ & \neg(i' < N) \wedge (\forall j.\, 0 \le j < N \implies A[j] = B[j]).\end{aligned}$$

Our technique addresses *deterministic* programs. A non-deterministic program in our context has an initial state that can both satisfy and violate the postcondition. More formally,

Definition 8 (Non-deterministic Modulo Postcondition CHCs). *Let S be a system of CHCs that has* **pre** *and extendable by a formula φ. We call S non-deterministic modulo postcondition if there exists an uniquely satisfiable formula s for which there are at least two satisfiable unrollings $\pi_{\langle C_0,\ldots,C_\ell\rangle}$ and $\overline{\pi}_{\langle C_0,\ldots,C_m\rangle}$ corresponding to extensions S_s and $\overline{S}_s$, respectively. Otherwise, we say S is deterministic*[6]

[6] An example is presented in [49].

We assume that the CHCs are representing terminating programs. Hence, for any initial state of a program encoded in CHCs, there exists an unrolling either satisfying or violating the postcondition.

Definition 9 (Terminating CHCs). *Let S be a system of CHCs with* **pre** *and extendable by a formula φ. We say S is terminating if there does not exist an infinite-length unrolling for $S_\top$ and $\overline{S}_\top$ (i.e. S and $\overline{S}$ are extended by $\varphi = \top$).*

3.3 Linear Array Loop Programs

Though our algorithms work at the level of CHCs, we are motivated to target CHCs representing *linear array loop programs* (or "linear loops" in short) that model real-world programs in existing array program verification works [9,39,10]. These are terminating programs with non-nested loops. We now present the syntax of a linear loop.

$$\begin{aligned}
\textit{program} &\rightarrow \texttt{assume}(\textit{pre}(V, A));\ \textit{stmts};\ \textit{post};\\
\textit{stmts} &\rightarrow \textit{assign}\ \big|\ \textit{forloop}\ \big|\ \textit{stmts}; \textit{stmts}\\
\textit{assign} &\rightarrow v = f(V, A)\ \big|\ a[i] = f(V, A)\ \big|\ \texttt{if}(\phi(V))\ \{\textit{assign}\}\ \texttt{else}\ \{\textit{assign}\}\\
&\quad\ \big|\ \textit{assign}; \textit{assign}\\
\textit{forloop} &\rightarrow \texttt{for}\ (i = l(V);\ c(i, V);\ i = h(i))\ \{\textit{assign}\}\\
\textit{post} &\rightarrow \texttt{assert}(\forall x.\, R(x, V) \implies Q(x, V, A))\ \big|\ \texttt{assert}(\exists x.\, R(x, V) \wedge Q(x, V, A))
\end{aligned}$$

Here V and A are disjoint sets of integer and array variables, respectively, $i \in V$ is a loop counter, $v \neq i \in V$ is an integer variable, f is a term over V and A such that any access to A is done by i, l is an integer term over V, h is an integer term over i which results in a monotonically increasing (or decreasing) assignment, and c is a guard of the form $i < u$ or $i > u$ for some integer term u over V, and ϕ is a boolean predicate. The postcondition ρ is given as a condition in `assert`, where R is a predicate in LIA over quantified and integer variables that represent a range of array elements, and Q is a property over an array with array read-access done only by x. For example, the formula $\forall x.\, 0 \leq x < N \implies B[x] = 42$ is in this form, where B is an array variable.

The precondition (and inductive invariants) inferred by our algorithm will be of the same quantification as the postcondition. Further, it can be conjunctions in case of universal quantification and disjunctions in case of existential quantification. Specifically, we consider preconditions (and inductive invariants) of the form described in (4). Such a form has been found effective in inferring inductive invariants in the existing works for array programs like [38,33,32,22].

$$\bigwedge \big(\forall x.\, R(x, V) \implies Q(x, V, A)\big) \quad \text{or} \quad \bigvee \big(\exists x.\, R(x, V) \wedge Q(x, V, A)\big) \tag{4}$$

A formal description of CHCs that represent linear loop programs is given in Sect. 6.1.

4 Undecidability of Maximal Precondition Inference for Linear Loops

Although linear loops and postconditions have syntactic restrictions, inference of maximal preconditions for such programs in the considered form (i.e. (4)) is still undecidable. In this section we prove this result.

We reduce the halting problem of *two-counter* machines [42] to the maximal precondition inference problem. Recall that a two counter machine $M = (C_1, C_2, L)$ has two counters C_1 and C_2, which are initially set to 0, and a finite set of instructions $L = \{l_1, \ldots, l_k\}$, where each instruction l_i is of type `inc`, `decjz`, and a designated `halt` instruction l_h . Given a two-counter machine $M = (C_1, C_2, L)$, deciding whether it halts, i.e. the halt instruction $l_h \in L$ is reached, is undecidable.

Theorem 1. *The problem of computing the maximal precondition for linear array loop programs in the form described in* (4) *is uncomputable.*

Proof Sketch We construct a linear array loop program with a single loop whose body simulates the execution of one transition of a two-counter machine, and an array records the locations the machine can reach after the transition.[7]

The undecidability of the problem notwithstanding, many real-life programs, like industrial battery controllers [9], adhere to linear array loop structures. Consequently, techniques like [39,10] have been developed to address such programs, but focusing on assertion checking rather than precondition inference. The existing precondition inference technique [54] finds it challenging to infer a precondition for such programs (details in Section 8). Motivated by these challenges, we propose a sound technique that infers maximal preconditions.

5 Inferring Preconditions and Invariants by Abduction

In this section, we give an overview of our approach for abductive inference of preconditions and inductive invariants. We first explain its basic principles, and then demonstrate them on the running example.

5.1 Overview

We assume that the input system of CHCs S represents a precondition inference problem, i.e. it has no facts, a single query, and a designated relation (***pre*** or ***cpre***) for the precondition. Since we are interested in the precondition inference for array programs, we assume that the query has a quantified constraint ρ.

The high-level algorithm is given in Algorithm 1. It is called InferAbd and is inspired by an earlier work on specification synthesis [50]. InferAbd incrementally attempts to discover an interpretation for each uninterpreted predicate

[7] All proofs are in [49].

Algorithm 1: InferAbd($S, \mathcal{M}, R$)

```
Input: $S$ – set of CHCs over $\mathcal{R}$, $R \subseteq \mathcal{R}$ – current subset (initially empty) of
       relations with invariants/preconditions, $\mathcal{M}$ – mapping from $\mathcal{R}$ to
       predicates, initially $\lambda r.\top$
Output: $\mathcal{M}$ – invariants/preconditions of $S$
1  if $R = \varnothing$ then
2      $R \leftarrow \{r \mid \exists s.\, rel(body(s)) = r \land rel(head(s)) = \bot \land s \in S\}$
3  $Worklist \leftarrow \{C \mid C \in S \land rel(body(C)) \in R\}$;
4  while $\exists C \in Worklist.$ CheckSAT($\neg\big(body(C) \implies head(C)\big)[\mathcal{M}/\mathcal{R}]$) do
5      let $\varphi$ be $\big(body(C) \implies head(C)\big)[\mathcal{M}/\mathcal{R}]$;
6      $\mathcal{M}(rel(body(C))) \leftarrow \mathcal{M}(rel(body(C))) \land$ Abduce($\varphi, args(body(C)), S$);
7      $\mathcal{M}(rel(body(C))) \leftarrow$Houdini($S, \mathcal{M}, R$);
8  if No $\mathcal{M}(\cdot)$ was strengthened or weakened then
9      if $R = \mathcal{R}$ then return $\mathcal{M}$;
10 $R \leftarrow \{r \mid \exists C \in S.\, rel(body(C)) = r \land rel(head(C)) \in R\}$;
11 return InferAbd($S, \mathcal{M}, R$);
```

in $\mathcal{R}$ by propagating the assertion backward, strengthening it when needed to establish inductiveness, or weakening if something went wrong during the inference of inductive invariants.

InferAbd (Algorithm 1) constructs a solution $\mathcal{M}$ for a system of CHCs recursively. $\mathcal{M}$ initially maps all the predicates in $\mathcal{R}$ to $\top$. At each call, the algorithm searches for a CHC C (line 3) such that $\mathcal{M}$ is not inductive at C. This inductiveness check is reduced to a satisfiability check, which is performed by an SMT solver (line 4). If φ is satisfiable then $\mathcal{M}$ is not inductive at the corresponding C, and thus $\mathcal{M}$ needs strengthening.

Note that in the first call of InferAbd, the initial $\mathcal{M}$ is inductive for all the CHCs except the query, thus the interpretations will be created for the predicates that appear in the body of the query. In the subsequent calls, these interpretations could be either strengthened or propagated through the bodies of the CHCs where they appear in the heads, towards the precondition.

In InferAbd, we write $\psi \leftarrow$ Abduce($\varphi, \vec{x}, S$) to denote an invocation of a new abduction algorithm (Algorithm 2) to obtain a formula ψ over variables $\vec{x}$ that makes φ valid. InferAbd uses Abduce as existing abduction solvers have limited support for arrays. In order to support arrays and quantifiers, Abduce abstracts quantified formulas over arrays and integers into quantifier-free formulas only over integers. To do this, Abduce considers two abduction queries for a CHC in S: 1) for the array element that is being rewritten (if any), and 2) for all other elements that are not changed. The formal description of Abduce is in Sect 6 along with illustration. However, by doing this "arrays-to-integer" reduction, Abduce could introduce some imprecision, which is fixed by running the Houdini algorithm (details in Sect 6.3).

InferAbd may not terminate because the series of strengthening predicates obtained in each iteration may diverge. But the recursion in InferAbd can be easily augmented by a threshold condition that forces the termination with an UNKNOWN result after reaching a predetermined recursion depth.

Theorem 2. *Whenever Algorithm 1 terminates, it returns a solution $\mathcal{M}$ to S.*

5.2 Approach in Action

We demonstrate the precondition inference approach on the example from Sect. 2 and Fig. 3.

Synthesizing an invariant for ***inv*** The algorithm begins with obtaining an initial candidate interpretation to ***inv*** from the query CHC. The predicate is the query constraint (i.e. the postcondition $\neg\rho$) with a slight modification:

$$\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda i,N,A,B,C.\, \forall j (0 \leq j < N \land j < i \implies A[j] = B[j]).$$

The modification includes dropping the loop condition and strengthening it by conjuncting a *range formula* [21] to the antecedent ($j < i$ here). In simple terms, the range formula is a predicate that represents the boundary between indices that are modified and not modified. It can be ($j < i$) or ($j > i$), based on whether the loop counter is increasing or decreasing, respectively (formal definition in 11).

Our algorithm then checks if any of the CHCs in *Worklist* are not valid. In this case, the second CHC is not valid. The algorithm then follows backward reasoning and attempts to update the current interpretation of ***inv*** by *abductive strengthening* to make it inductive using a series of SMT checks and quantifier elimination queries.

The algorithm does abductive strengthening by posing two queries. The first one is to accommodate the write to the i-th element of the array. This strengthening for the second CHC is posed as an abduction query for ψ_1 that is constructed by restricting to only a single cell of the array that is rewritten in the loop:

$$\psi_1(A, B, C, j) \land A'[j] = A[j] \land B'[j] = B[j] \land \boldsymbol{C'[j] = j} \implies A'[j] = B'[j].$$

Here, all the array terms (like $A[j]$, $A'[j]$, $B[j]$, etc.) are further replaced by fresh integer variables which allows us to use a standard abduction solver and get the following solution:

$$\psi_1 \mapsto \lambda A, B, C, j.\, A[j] = B[j].$$

Intuitively, ψ_1 gives the *weakest precondition* on $A[i]$, $B[i]$ and $C[i]$ before the i-th loop iteration, such that the desired postcondition holds for $A'[i]$ and $B'[i]$ after the iteration.

The second abduction query accommodates all the other elements in the range $0 \leq j < N \land j \neq i$ that are unaffected in the i-th iteration:

$$\psi_2(A, B, C, j) \land A'[j] = A[j] \land \boldsymbol{C'[j] = C[j]} \land B'[j] = B[j] \implies A'[j] = B'[j].$$

The delta w.r.t. the first query is shown in bold. This query also has the same solution as ψ_1.

$$\psi_2 \mapsto \lambda A, B, C, j.\, A[j]=B[j].$$

To build the new invariant from ψ_2 and ψ_1 to the new invariant candidate, we split the array range into two segments based on the range formula, and its negation:

$$\begin{aligned}\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda A, B, C, i. \forall j (0 \le j < N \wedge j < i \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j < i \wedge 2 * j < N \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j \ge i \wedge 2 * j < N \implies A[j]=B[j]).\end{aligned}$$

The second conjunct is derived from ψ_2 and the range formula ($j < i$), whereas the third conjunct is from ψ_1 and negation of the range formula ($j \ge i$). If the CHC has any additional constraints (like $2 * i < N$ here) that will be added in the antecedent as well.

While validating this candidate, the algorithm goes over the CHCs again and checks the implications: it now turns out to be not inductive for the third CHC. The algorithm thus repeats the abductive strengthening and poses two queries:

$$\begin{aligned}\psi_3(A, B, C, j) \wedge B'[j]=B[j] \wedge C'[j]=C[j] \wedge \boldsymbol{A'[j]=C[j]} \implies A'[j]=B'[j], \\ \psi_4(A, B, C, j) \wedge B'[j]=B[j] \wedge C'[j]=C[j] \wedge \boldsymbol{A'[j]=A[j]} \implies A'[j]=B'[j],\end{aligned}$$

getting the following next candidate, that is subsequently validated:

$$\begin{aligned}\boldsymbol{inv} \mapsto \lambda A, B, C, i. \forall j (0 \le j < N \wedge j < i \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j < i \wedge 2 * j < N \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j \ge i \wedge 2 * j < N \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j < i \wedge 2 * j \ge N \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge j \ge i \wedge 2 * j \ge N \implies B[j]=C[j]).\end{aligned}$$

Synthesizing $\boldsymbol{pre}$ Finally, the precondition is obtained from the solution for $\boldsymbol{inv}$. Because the first CHC initializes the counter i to zero, all the conjuncts with $j < i$ simplify to true and the rest simplifies to:

$$\begin{aligned}\boldsymbol{pre} \mapsto \lambda N, A, B, C. \forall j (0 \le j < N \wedge 2 * j < N \implies A[j]=B[j]) \wedge \\ \forall j (0 \le j < N \wedge 2 * j \ge N \implies B[j]=C[j]).\end{aligned}$$

6 Range Abduction

In this section, we present our technique called *range abduction* for inferring quantified invariants, and subsequently, quantified preconditions. We define the ABDUCE method for quantified formulas over arrays and linear arithmetic that can be used in the general algorithm of abductive invariant synthesis. Its core features include the capability to selectively apply quantifier elimination, such that it keeps all quantifiers that are explicit in the abducible formula. As its main computational vehicle, the method uses quantifier elimination over linear arithmetic on formulas produced from the actual abducibles by over-approximating (as precisely as possible) the array computation.

6.1 Preliminaries

CHCs We first formally describe the CHC structure that we support corresponding to linear loops. We assume that the inputs are given as CHCs, where bodies are in CNF (otherwise, it can be transformed following Sect. 3.2). For each CHC, we consider two disjoint vectors of source (resp., destination) variables, $\vec{v}$ and $\vec{a}$ (resp., $\vec{v}\,'$ and $\vec{a}\,'$), such that only $\vec{a}$ (resp., $\vec{a}\,'$) consists of array variables.

We allow only a single index to access elements of all arrays $b \in \vec{a}$ in each CHC C, and without loss of generality we assume that it is an integer variable $i \in \vec{v}$ (usually, a loop counter).[8] For simplicity, we also introduce a set of temporary integer variables $\vec{t}$ that store some elements selected from arrays and can be used in other parts of C (e.g., to compute the next value to be written to an array b' via some function f). Thus, we assume that only three possible types of constraints are used to equate arrays (or their elements), and that they appear in recursive CHCs, that is:

$$\begin{aligned} &\boldsymbol{inv}_1(\vec{v},\vec{a}) \land \big[(a' = a\land)^*\big] \land \big[(t = a[i]\land)^*\big]\land \\ &\qquad \big[(b' = \mathit{store}(b,i,f(\vec{v},\vec{t}))\land)^*\big] \land \varphi(\vec{v},\vec{v}\,',\vec{t}) \implies \boldsymbol{inv}_2(\vec{v}\,',\vec{a}\,') \end{aligned} \tag{5}$$

where * is Kleene star, $a, b \in \vec{a}$, $a', b' \in \vec{a}\,'$, $t \in \vec{t}$, and φ is over only non-array variables. Note that sequences of stores (e.g., nested) could be supported after some sort of a CHC normalization, e.g., by introducing temporary uninterpreted predicates and splitting C. Symbols $\boldsymbol{inv}_1$ and $\boldsymbol{inv}_2$ might refer to the same predicate.

Queries There is only a single query among CHCs, and it has the form of either of the two implications:

$$\boldsymbol{inv}(\vec{v},\vec{a})\land\varphi(\vec{v}) \land \exists x.(R(x,\vec{v}) \land Q(x,\vec{v},\vec{a})) \implies \bot \tag{6}$$

$$\boldsymbol{inv}(\vec{v},\vec{a})\land\varphi(\vec{v}) \land \forall x.(R(x,\vec{v}) \implies Q(x,\vec{v},\vec{a})) \implies \bot \tag{7}$$

In the body of the query, there is a quantifier-free conjunct φ and a quantified formula with subformulas R and Q. Formula φ could represent the termination condition of the array processing loop/recursion (captured in the other CHCs). The subformulas R and Q could represent, respectively, a range of elements in

[8] In practice, the restrictions about array accesses and the shape of the CHC can be relaxed, but requires a more careful handling than we propose in this paper. Our implementation has it, but the paper omits it to maintain the simplicity of presentation.

an array (giving a condition over possible index x of the array), and a property over an array element (indexed using the x variable). We restrict read-accesses of arrays to the quantified variable only.

The formula in the query determines an initial candidate interpretation for the predicate in the query. For instance, in (6) and (7), respectively:

$$\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda \vec{v}, \vec{a}.\, \forall x.\, (R(x, \vec{v}) \implies \neg Q(x, \vec{v}, \vec{a})) \tag{8}$$

$$\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda \vec{v}, \vec{a}.\, \exists x.\, (R(x, \vec{v}) \wedge \neg Q(x, \vec{v}, \vec{a})) \tag{9}$$

Applying Algorithm 1 We assume that an iteration of the algorithm deals with a mapping $\mathcal{M}$ and the following CHC, where $\mathcal{M}(\boldsymbol{inv}_1)$ might be currently $\top$, but $\mathcal{M}(\boldsymbol{inv}_2)$ is quantified:

$$\boldsymbol{inv}_1(\vec{v}, \vec{a}) \wedge \varphi(\vec{v}, \vec{a}, \vec{v}\,', \vec{a}\,') \implies \boldsymbol{inv}_2(\vec{v}, \vec{a}) \tag{10}$$

Abductive strengthening is needed when the following implication is not valid on substitutions of interpretations of $\boldsymbol{inv}_1$ and $\boldsymbol{inv}_2$ (line 6 of Algorithm 1), thus necessitating to find ψ, such that the following is valid:

$$\psi \wedge \varphi \implies \mathcal{M}(\boldsymbol{inv}_2) \tag{11}$$

Intuitively, for ψ our algorithm reuses the quantified structure of $\mathcal{M}(\boldsymbol{inv}_2)$. For all quantifier-free conjuncts of $\mathcal{M}(\boldsymbol{inv}_2)$, strengthening is done following the simple abduction, like e.g., in [17]. For quantified formulas, the algorithm is trickier. In the rest of this section, we assume that algorithms are strengthening w.r.t. formulas $\pi_\exists$ and $\pi_\forall$ having the forms, respectively (9) and (8).

6.2 Core Technique

The basic principle behind our quantified abductive strengthening is in the preservation of the range. That is, if the quantified formula on the right side of (11) has form (9) or (8), then it intuitively means that some property $Q(x, \vec{v}, \vec{a})$ should hold either for all elements of array(s) (when quantification is universal), or some elements of arrays $\vec{a}$ (when quantification is existential), determined by $R(x, \vec{v})$. Thus, an interpretation of predicate ψ on the left side of (11) should also constrain all elements of (some) arrays belonging to the same range.

Since by our syntax restrictions we allow elements of arrays $b \in \vec{a}$ to be rewritten using only a single index i, each constraint $b' = \mathit{store}(b, i, f(\vec{v}, \vec{t}))$ can be safely replaced in the CHC body as:

$$b'[i] = f(\vec{v}, \vec{t}) \qquad \text{and} \qquad \forall j.\, i \neq j \implies b'[i] = b[i]$$

In the following, we are going to use the auxiliary mapping to reduce abduction over array and integer variables to purely integer abduction.

Definition 10. *Let $\vec{a}$, $\vec{t}$, and $\vec{t'}$ be sets of array variables, integer variables, and integer terms, respectively, all of the same cardinality. A bijection $\mathcal{SS} : \vec{t'} \to \vec{t}$ is called* select-substitution *w.r.t. index i, if for every $a \in \vec{a}$, there exists $t \in \vec{t}$ such that $\mathcal{SS}(a[i]) = t$.*

Algorithm 2: Abduce($\pi, \vec{x}, S$)

```
Input: $\pi$ – abducible formula of the form (11) from a CHC $C$, $\vec{x}$ – variables
       to keep, $S$ – set of CHCs over $\mathcal{R}$ where $C \in S$, $\mathcal{M}$ – mapping from $\mathcal{R}$
       to predicates
Output: $\psi$ a strengthening for $\pi$
1  $\langle \pi_1, \pi_2 \rangle \leftarrow$ decompose $\pi$ into (12) and (13);
2  for $k \in [1, 2]$ do
3      $\pi_k \leftarrow$ unquantify and apply some $\mathcal{SS}$ to $\pi_k$;
4      $\psi'_k \leftarrow$ solve integer abduction for $\pi_k$;
5      $\psi_k \leftarrow$ apply $\mathcal{SS}^{-1}$ to $\psi'_k$ and replace $i$ by $x$;
6  $\sigma \leftarrow$ ComputeRangeFormula($S$);
7  $\theta \leftarrow$ GetCondition($C$);
8  if $\pi$ universally quantified then
9      $\psi_1 \leftarrow \psi_1 \vee \sigma \vee \neg\theta$;
10     $\psi_2 \leftarrow \psi_2 \vee \neg\sigma \vee \neg\theta$;
11     $\psi \leftarrow \forall x.\, \psi_1 \wedge \forall x.\, \psi_2$;
12 if $\pi$ existentially quantified then
13     $\psi_1 \leftarrow \psi_1 \wedge \neg\sigma \wedge \theta$;
14     $\psi_2 \leftarrow \psi_2 \wedge \sigma \wedge \theta$;
15     $\psi \leftarrow \exists x.\, \psi_1 \vee \exists x.\, \psi_2$;
16 return $\psi$;
```

The pseudocode of our range abduction is given in Algorithm 2. Below we discuss its details.

Universally-quantified formulas (8) The abduction query π of the form (11) can be decomposed (line 1) into two stronger abduction queries, π_1, π_2:

$$\psi_1 \wedge \left[(b'[i] = f(\vec{v}, \vec{t}) \wedge)^* \right] .. \implies (R(i, \vec{v}) \implies Q(i, \vec{v}, \vec{a})) \tag{12}$$

$$\psi_2 \wedge \left[(b'[i] = b[i] \wedge)^* \right] ... \implies (R(i, \vec{v}) \implies Q(i, \vec{v}, \vec{a})) \tag{13}$$

Since $\mathcal{M}(\boldsymbol{inv}_2)$ is universally-quantified and due to our syntactic restrictions, only the i-th elements of any source arrays are relevant for the abduction query. Thus, without loss of generality, our algorithm lowers the (possibly) universally-quantified formula in $\mathcal{M}(\boldsymbol{inv}_2)$ to a quantifier-free formula over the i-th array element, and further replaces all the array access terms of the form $a[i]$ to integer terms a_i using a select-substitution $\mathcal{SS}$, essentially boiling down to two abduction queries over pure integer arithmetic with abducibles ψ'_1 and ψ'_2 (lines 3, 4).

After the abduction solver returns ψ'_1 and ψ'_2 for the integer arithmetic queries, the $\mathcal{SS}^{-1}$ mapping is applied to replace integer terms a_i by array terms $a[i]$ to get ψ_1 and ψ_2 that constitute solutions to queries (12) and (13)(line 5).

It remains finally to re-introduce the universal quantifier for x to $\psi_1[x/i]$ and $\psi_2[x/i]$ to get a solution to our main abduction query (11). There are several ways to do it. One way is to not introduce quantifiers for ψ_1 as the query (12) captures the effect of a single store to an i-th element of an array. For ψ_2, then, the quantifier's range will span over all the original range except i. However, this way, seemingly obvious, does not work in practice because the produced invariant is unlikely to be inductive.

Another way is to split the range into two segments with the border at i. It would intuitively correspond to the range formula computation of [22], i.e., the sub-array that has already been processed in the loop encoded by the CHC, and the sub-array that remains to be processed. The former restricts the range of ψ_2 (lines 10, 14) and the latter of ψ_1 (lines 9, 13). More formally:

Definition 11. *For an inductive CHC C with loop counter i, where i is in the interval $[l, u]$, and a free variable j, the range formula is $j < i$ when $i \geq l$ is inductive at C, and $j > i$ when $i \leq u$ is inductive at C.*

In Algorithm 2, σ is the range formula returned by ComputeRangeFormula. Additionally, GetCondition adds predicates that are present in the constraint of the CHC (like $2 * i < N$) after substituting the loop counters in them by the quantified variables.

Existentially-quantified formulas (9) Similar to the universally-quantified case, the abduction query (11) for existential quantification will be decomposed into two abduction queries. Queries (12) and (13) in this case have the form:

$$\begin{aligned} \psi_1 \wedge \left[(b'[i] = f(\vec{v}, \vec{t})\wedge)^*\right]\ldots &\implies (R(i, \vec{v}) \wedge Q(i, \vec{v}, \vec{a})) \\ \psi_2 \wedge \left[(b'[i] = b[i]\wedge)^*\right]\ldots &\implies (R(i, \vec{v}) \wedge Q(i, \vec{v}, \vec{a})) \end{aligned}$$

The remainder of the algorithm in this case is the same as in the universally-quantified case with the exception that we *disjoin* two quantified solutions for the abduction queries before checking if it is inductive.

6.3 Houdini Algorithm

The strengthening performed by Algorithm 2 might result in a too strong candidate invariant for already validated CHCs. To resolve this, Algorithm 1 weakens the candidate invariants by using an existing algorithm called Houdini [24](line 7). Given a set of relations R and a mapping $\mathcal{M}$, Houdini recursively weakens $\mathcal{M}$ until it is inductive at each CHC C whose $rel(head(C)) \in R$. It does this by finding a counterexample to inductiveness and dropping the conjuncts that don't satisfy the counterexample.

6.4 Illustration of Existentially Quantified Precondition Inference

We end this section by illustrating Algorithm 1 on an existentially quantified postcondition from Fig. 4. The CHCs of this program are given in Fig. 5.

The algorithm chooses an initial candidate for ***inv*** from the query. The loop condition is dropped like universal quantification, but the range formula is not

$$\begin{aligned}
\boldsymbol{cpre}(N,A,B,C) \wedge i=0 &\Longrightarrow \boldsymbol{inv}(i,N,A,B,C)\\
\boldsymbol{inv}(i,N,A,B,C) \wedge i<N \wedge 2*i<N \wedge C'=\mathit{store}(C,i,i) \wedge i'=i+1 &\Longrightarrow \boldsymbol{inv}(i',N,A,B,C')\\
\boldsymbol{inv}(i,N,A,B,C) \wedge i<N \wedge 2*i\geq N \wedge A'=\mathit{store}(A,i,C[i]) \wedge i'=i+1 &\Longrightarrow \boldsymbol{inv}(i',N,A',B,C)\\
\boldsymbol{inv}(i,N,A,B,C) \wedge \neg(i<N) \wedge \neg(\exists j.\, 0\leq j<N \wedge A[j]\neq B[j]) &\Longrightarrow \bot
\end{aligned}$$

Fig. 5: CHC encoding of program in Fig. 4.

conjuncted for existential postcondition as this often results in a too strong precondition, viz. $\bot$.

$$\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda i,N,A,B,C.\, \exists j.\, 0\leq j<N \wedge A[j]\neq B[j]$$

Algorithm 1 now checks if either the second or third CHC in the *Worklist* is not inductive. Since the third CHC is not inductive, Abduce is called. The result of two abduction queries corresponding to i-th element and non i-th element, i.e. ψ_1 and ψ_2, will be $B[j]\neq C[j]$ and $A[j]\neq B[j]$, respectively. Further, quantification and range formulas are added, which will result in the candidate:

$$\begin{aligned}
\boldsymbol{inv} \overset{\text{cand}}{\mapsto} \lambda i,N,A,B,C.\, &\exists j.\, 0\leq j<N \wedge A[j]\neq B[j]\wedge\\
&\big(\exists j.\, 0\leq j<N \wedge j\geq i \wedge 2*j\geq N \wedge B[j]\neq C[j]\vee\\
&\exists j.\, 0\leq j<N \wedge j<i \wedge 2*j\geq N \wedge A[j]\neq B[j]\big)
\end{aligned}$$

Now, the Houdini algorithm finds that the candidate is not inductive at the third CHC. For instance, it finds a counterexample to validity of the form:

$$\begin{aligned}
a[j]\neq b[j] \text{ for } j=i \text{ , otherwise } a[j]=b[j]\\
b[j]\neq c[j] \text{ for } j=i+2 \text{ , otherwise } b[j]=c[j]
\end{aligned}$$

It drops the conjunct $\exists j.\, 0\leq j<N \wedge A[j]\neq B[j]$ that does not satisfy the counterexample. The rest are found to be inductive at the third and second CHCs.

$$\begin{aligned}
\boldsymbol{inv} \mapsto \lambda i,N,A,B,C.\, &\exists j.\, 0\leq j<N \wedge j\geq i \wedge 2*j\geq N \wedge B[j]\neq C[j]\vee\\
&\exists j.\, 0\leq j<N \wedge j<i \wedge 2*j\geq N \wedge A[j]\neq B[j]
\end{aligned}$$

Finally, the precondition $\boldsymbol{cpre}$ is computed from the first CHC by substituting $i=0$, resulting in:

$$\boldsymbol{cpre} \mapsto \lambda i,N,A,B,C.\, \exists j.\, 0\leq j<N \wedge 2*j\geq N \wedge B[j]\neq C[j]$$

7 Maximal Preconditions

The interpretation of $\boldsymbol{pre}$ generated by Algorithm 1 is guaranteed to be a precondition by Theorem 2, but it could be non-maximal. That is, it may exclude

Algorithm 3: MaximalPrecond($S, \boldsymbol{pre}$)

```
Input: $S$ – set of CHCs over $\mathcal{R}$, $\boldsymbol{pre} \in \mathcal{R}$ – precondition relation
Output: $\mathcal{M}(\boldsymbol{pre})$ – Maximal precondition for $\boldsymbol{pre}$
1  $\mathcal{M} \leftarrow$ InferAbd($S, \{\lambda r \in \mathcal{R}. \top\}$);
2  $\overline{\mathcal{M}} \leftarrow$ InferAbd($\overline{S}, \{\lambda r \in \mathcal{R}. \top\}$);
3  $\phi \leftarrow \neg(\neg\mathcal{M}(\boldsymbol{pre}) \implies \overline{\mathcal{M}}(\boldsymbol{cpre}))$;
4  while CheckSAT($\phi$) do
5      $ctm \leftarrow$ GetModel($\phi$);
       // $ctm$ is of the form $\bigwedge_{0 \leq i \leq n} x_i = c_i$
6      $postViolated \leftarrow$ UnrollCHC($S_{ctm}, \overline{S}_{ctm}$);
7      if $postViolated$ then
8          $\overline{\mathcal{M}} \leftarrow$ Weaken($\overline{S}, \overline{\mathcal{M}}(\boldsymbol{cpre}) \vee ctm$)
9      else
10         $\mathcal{M} \leftarrow$ Weaken($S, \mathcal{M}(\boldsymbol{pre}) \vee ctm$)
11     $\phi \leftarrow \neg(\neg\mathcal{M}(\boldsymbol{pre}) \implies \overline{\mathcal{M}}(\boldsymbol{cpre}))$;
12 return $\mathcal{M}(\boldsymbol{pre})$;
```

some initial states from which the postcondition holds. In this section, we propose a technique that checks whether a precondition is maximal (i.e. logically weakest). If not, it incrementally weakens the precondition in a loop until it becomes maximal.

7.1 Overview

Algorithm 3 gives a description of the maximality checker. Given a precondition inference problem via a system of CHCs S, it returns a maximal precondition on termination. It first generates a precondition for S using Algorithm 1. In order to check whether the precondition is maximal, the algorithm infers another precondition for the complement CHC system $\overline{S}$ (line 2). Recall from Definition 5 that this system has the same structure as S except the postcondition in the query is complemented. To avoid confusion, we consider $\boldsymbol{pre}$ of this system is substituted by another uninterpreted relation with the same arity $\boldsymbol{cpre}$. For example, Fig 5 is the complement CHC system of Fig 3.

The maximality check is performed next by checking whether all the states that are outside $\mathcal{M}(\boldsymbol{pre})$ are in $\overline{\mathcal{M}}(\boldsymbol{cpre})$(line 4). Intuitively, if all the states in $\neg\mathcal{M}(\boldsymbol{pre})$ are in $\overline{\mathcal{M}}(\boldsymbol{cpre})$ then those states violate the postcondition as $\overline{\mathcal{M}}(\boldsymbol{cpre})$ is the precondition of the complement postcondition. The validity check is reduced to a satisfiability check by negation and the model to the satisfiability check is called a *counterexample-to-maximality*, or *CTM*.

The algorithm uses the CTM to determine which of $\boldsymbol{pre}$ or $\boldsymbol{cpre}$ has to be weakened by invoking the method UnrollCHC (line 6). Intuitively, UnrollCHC performs a task similar to executing the program represented by CHCs with *CTM* as the initial state. More precisely, UnrollCHC will find unrollings (Definition 7) of different lengths for the extensions S_{ctm} and $\overline{S}_{ctm}$ and terminates when an unrolling is satisfiable. It then returns whether the unrolling was from S_{ctm}, or $\overline{S}_{ctm}$. For a deterministic CHC system (Definition 8), a satisfiable unrolling exists either for S_{ctm}, or $\overline{S}_{ctm}$.

In the next step, the algorithm will weaken $\boldsymbol{cpre}$ if the unrolling is from S_{ctm}, or $\boldsymbol{pre}$ if the unrolling is from $\overline{S}_{ctm}$. The weakening is performed by WEAKEN, which will be called with an appropriate CHC system and the current interpretation for the precondition(lines 10, 8). WEAKEN will generalize the precondition and find inductive invariants. This loop of checking for CTM and weakening one of the precondition continues till the maximal precondition is found.

Theorem 3. *The precondition returned by Algorithm 3, when it terminates, is maximal when S is deterministic and terminating.*

Example 3. In Sect 5.2 and Sect 6.4, Algorithm 1 found the following interpretations for $\boldsymbol{pre}$ and $\boldsymbol{cpre}$:

$$\begin{aligned}\boldsymbol{pre} \mapsto \lambda N, A, B, C.\, &\forall j.\, 0 \leq j < N \wedge 2 * j < N \Longrightarrow A[j] = B[j] \wedge \\ &\forall j.\, 0 \leq j < N \wedge 2 * j \geq N \Longrightarrow B[j] = C[j].\end{aligned}$$

$$\boldsymbol{cpre} \mapsto \lambda i,N,A,B,C.\, \exists j.\, 0 \leq j < N \wedge 2 * j \geq N \wedge B[j] \neq C[j].$$

The reader may notice that $\boldsymbol{cpre}$ is not maximal, hence it is not possible to check whether $\boldsymbol{pre}$ is maximal. We now illustrate how Algorithm 3 determines this.

After finding the interpretations, Algorithm 3 checks the following formula:

$$\begin{gathered}\neg(\forall j.\, 0 \leq j < N \wedge 2 * j < N \Longrightarrow A[j] = B[j] \wedge \forall j.\, 0 \leq j < N \wedge 2 * j \geq N \Longrightarrow B[j] = C[j]) \\ \Longrightarrow \\ \exists j.\, 0 \leq j < N \wedge 2 * j \geq N \Longrightarrow B[j] \neq C[j]\end{gathered}$$

Since this formula is satisfiable, the algorithm deduces that at least one among $\mathcal{M}(\boldsymbol{pre})$ and $\mathcal{M}_c(\boldsymbol{cpre})$ is not maximal. Suppose it gets the following satisfiability model, or CTM:

$$N = 1 \wedge A[0] = 0 \wedge B[0] = 1 \wedge C[0] = 0.$$

UNROLLCHC finds that the CHCs violate the property when the CTM is the initial state. Hence, $\boldsymbol{cpre}$, the precondition of negation of the property, can be weakened by at least one point, viz. CTM.

Algorithm 4: $\textsc{Weaken}(S,\ \mathcal{M}(\boldsymbol{pre}) \vee ctm)$

```
Input: $S$ – set of CHCs over $\mathcal{R}$, $\mathcal{M}(\boldsymbol{pre}) \vee ctm$
Output: $\mathcal{M}'$ – a solution to $S$ with $\mathcal{M}(\boldsymbol{pre}) \vee ctm \implies \mathcal{M}'(\boldsymbol{pre})$
1 $G \leftarrow \textsc{ConstructGrammar}(S, \mathcal{M}(\boldsymbol{pre}))$;
2 while $\top$ do
3     $\sigma \leftarrow \textsc{NextCandidate}(G)$;
4     if $\textsc{CheckSAT}(\neg(ctm \implies \sigma))$ then continue;
5     $\varphi \leftarrow \mathcal{M}(\boldsymbol{pre}) \vee \sigma$;
6     for $i \in [0 \cdots n]$ where $\mathcal{R} = \{r_0 = \boldsymbol{pre}, r_1 \cdots r_n\}$ do
7         $\mathcal{M}'(r_i) \leftarrow \textsc{InvInfer}(S, \mathcal{M}', r_i)$ or $\varphi$ for $r_0$;
8     if $\exists C \in S_\varphi.\, \textsc{CheckSAT}(\neg(body(C) \implies head(C))[\mathcal{M}'/\mathcal{R}])$ then
        continue;
9     return $\mathcal{M}'$;
```

7.2 Weakening of Precondition

Once the precondition that has to be weakened is determined, a trivial weakening is to add the CTM to the current interpretation. However, this may cause non-termination as there can be infinitely many CTMs. In this section, we propose a heuristic in Algorithm 4 that can accelerate the weakening process.

Algorithm 4 works in two stages. First, it finds a formula φ that is generally weaker than the trivial solution $\mathcal{M}(\boldsymbol{pre}) \vee ctm$ (lines 3- 5). To do this, it enumerates (line 3) a formula σ from an input grammar G (a sample grammar is given in [49]) and then checks if it is weaker. Then, it finds inductive invariants $\mathcal{M}'$ (line 7) for the extended system S_φ (recall Definition 6) using a slightly modified version of range abduction (algorithmic description is in [49]). By Lemma 2, φ and $\mathcal{M}'$ together forms a solution to the input system S.

Theorem 4. *Algorithm 4 returns a solution $\mathcal{M}'$ to S, and $\mathcal{M}(\boldsymbol{pre}) \vee ctm \implies \mathcal{M}'(\boldsymbol{pre})$*

Example 4. We continue illustration of Example 3. Algorithm 4 is called with a complement CHC system (Fig 5) and $\mathcal{M}(\boldsymbol{cpre}) \mapsto \lambda i,N,A,B,C.\, \exists j.\, 0 \le j < N \wedge 2*j \ge N \wedge B[j] \ne C[j]$ and $ctm = N = 1 \wedge A[0] = 0 \wedge B[0] = 1 \wedge C[j] = 0$. Suppose that the algorithm samples σ as $\exists j.\, 0 \le j < N \wedge 2*j < N \wedge A[j] \ne B[j]$ based on the constraints from query and second CHC. Since the check at line 4 passes, φ will be assigned:

$$\exists j.\, 0 \le j < N \wedge 2*j \ge N \wedge B[j] \ne C[j] \vee \exists j.\, 0 \le j < N \wedge 2*j < N \wedge A[j] \ne B[j].$$

$\textsc{InvInfer}$ uses the postcondition to compute $\mathcal{M}[r_{i+1}]$ and σ to compute $\mathcal{M}[r_{i-1}]$. It then adds $j < i$ to the former, $j \ge i$ to the latter, and disjuncts them (due to existential quantification) to get:

$$\begin{aligned} \boldsymbol{inv} \mapsto \lambda i,N,A,B,C.\, & \exists j.\, 0 \le j < i \wedge A[j] \ne B[j] \vee \\ & \exists j.\, i \le j < N \wedge 2*j \ge N \wedge B[j] \ne C[j] \vee \\ & \exists j.\, i \le j < N \wedge 2*j < N \wedge A[j] \ne B[j] \end{aligned}$$

Since this is inductive at all CHCs, the algorithm returns with φ and $\boldsymbol{inv}$. Algorithm 3 will perform its check and finds that $\boldsymbol{pre}$ is maximal.

8 Evaluation

Tool We implemented our algorithms in a tool called PreQSyn on top of the FreqHorn framework [22]. Our tool takes as input a precondition-inference problem encoded as a set of CHCs. It uses Z3 [44] to solve SMT queries. Quantifier elimination is performed using the solver from [20] that uses model-based projection [5]. On a successful execution, our tool infers maximal preconditions and inductive invariants for the loops.

Evaluation Goals We evaluate PreQSyn on the following research questions:

RQ1 Can PreQSyn infer universal and existential preconditions? How many of them can it prove maximal?
RQ2 Can PreQSyn compete with existing maximal quantified precondition inference tools?
RQ3 How challenging for state-of-the-art is to infer invariants *with* preconditions?
RQ4 How do various modules of PreQSyn influence its performance?

Benchmarks and Configuration We use 32 precondition inference problems with 29 universal and 3 existential quantified postconditions. Since none of the benchmarks from [54] had quantified postconditions, we derived a majority (26/32) of benchmarks from the existing *verification benchmarks* of [22] that have been collected from various sources like SV-COMP. In particular, we considered 48 benchmarks from the public repository of [22] that have multiple loops, i.e., the first loop has an array initialization, and the other loops involve various types of array processing like copying, modifying, filtering, and searching among the elements. We then excluded the first (initialization) loop from each benchmark, thus targeting the necessity of synthesizing a quantified precondition that would intuitively describe how the arrays need to be initialized in order to meet the postcondition. We further excluded benchmarks that gave repetitive problems (8/48) and did not meet our syntactic restrictions (viz. had non-quantified postconditions (6/48), had nested loops (5/48), or had non-linear expressions (3/48)). We added 6 new benchmarks to test different features of our tool.

We performed the experiments on an Ubuntu 20.04 machine with a 2.5 GHz processor and 16 GB memory. A timeout of 100s was given to all the tools.

RQ1 PreQSyn inferred a precondition for 31/32 benchmarks. The failed benchmark timed out in the inductiveness check. Out of 31 preconditions, 22 were proved to be maximal automatically. All the successful benchmarks were completed within 5 seconds. Overall, PreQSyn solved CHC tasks numbering 31 with universally quantified and 30 existentially quantified postconditions corresponding to $\boldsymbol{pre}$ and $\boldsymbol{cpre}$.

On manual inspection of 9 benchmarks for which PreQSyn found a precondition but was unable to prove maximality, 5 were found to be non-deterministic. However, the inferred preconditions for them were sufficiently weak. The rest 4 failed in different stages of weakening ***cpre***. Among these benchmarks, we found that 3/4 preconditions (i.e. ***pre***) were actually maximal.[9]

RQ2 We ran P-Gen (with Z3 v2.0 as its SMT solver) on semantically equivalent C programs manually constructed from the CHCs. P-Gen found only 2/32 preconditions as maximal. Both of them were existentially quantified. It timed out on 5/32 benchmarks. On the remaining 25/32 benchmarks it exited without finding a precondition. Overall, PreQSyn inferred significantly more preconditions than P-Gen due to the generalization capability of range abduction.

RQ3 We tried to replace our invariant inference technique by an existing one, thus evaluating the need to discover our invariants. Existing state-of-the-art CHC solving tools can handle arrays, to some extent, namely: Spacer [32](Z3 v4.8.10), a PDR-based invariant inference tool, and FreqHorn [22] (v.0.6), a SyGuS based invariant inference tool. So we pose the simpler problem of inferring invariants *with* preconditions to them. Furthermore, we also pose this as an assertion checking problem to VeriAbs [15] (v1.4.2), a portfolio solver that targets linear loops and the gold winner of SV-COMP 2022 ReachSafety Category [4] and the winner of array category since several years.

To create invariant inference and verification problems, we consider 42 precondition inference problems corresponding to ***pre*** and ***cpre*** for which PreQSyn was able to find the maximal preconditions. The 42 precondition inference problems were converted manually to verification problems by using the maximal preconditions. For Spacer, the CHCs were annotated by the maximal interpretations of ***pre*** and ***cpre***, for VeriAbs, semantically equivalent C programs with maximal preconditions as loops, and for FreqHorn original CHCs were provided as input.

Out of 42 problems, 21 each of universally and existentially quantified postconditions, VeriAbs solved 37, FreqHorn solved 20, and Spacer solved 11.

RQ4 We disabled Houdini algorithm from line 7 of Algorithm 1 and PreQSyn found preconditions for 27 benchmarks compared to 31 with the range abduction algorithm. Out of 27, only 6 were proved maximal. We conclude that weakening by Houdini is useful, especially when postconditions are existentially quantified. We extended the SMT-based maximality checking algorithm from [50], but it was unsuccessful in proving the 21 problems that our maximality checker proved.

9 Related Work

The problem of precondition inference appears in multiple applications and has been the subject of numerous works. Broadly, these works can be classified as

[9] Detailed results of evaluation with timings can be found in [49].

static [45,14,54,13], dynamic [53,25,3,41], and a mix of both [46]. Our technique falls in the first category. The two works closest to ours are [14] and [54] which compute maximal quantified preconditions for array programs, using abstract interpretation and CEGAR-based predicate abstraction, respectively. Unlike the technique in [14], our work does not require predefined abstract domains. The technique in [54] computes over-approximations of safe and unsafe states (i.e. over-approximations of $\boldsymbol{pre}$ and $\boldsymbol{cpre}$) and then refines them till they become disjoint. The over-approximations are computed using predicate abstraction and the predicates required for the refinement of the abstraction are derived from a set of heuristic rules. Our technique differs from theirs in several ways: we rely on abduction-based techniques to infer necessary predicates, while they rely on minimal unsat cores; we infer quantified inductive invariants that witness the correctness of the inferred preconditions, while their technique does not; finally, we target quantified postconditions while they consider only quantifier-free postconditions.

The problem of inferring universally quantified inductive invariants has received considerable attention. The inference is made using methods such as abstract interpretation [30], predicate abstraction using Skolem constants [40] and interpolation [34], an extension of IC3 for arrays [32], and syntax-guided synthesis [22]. These techniques, apart from being restricted to universal quantification, also expect a precondition. Our technique overcomes these limitations by inferring preconditions including existentially quantified ones.

Many techniques verify programs with arrays by transforming them to a sound abstraction without explicitly generating inductive invariants. The abstraction can be obtained by considering all the array elements as a single cell [7], or multiple fixed cells and then converting to array-free nonlinear CHCs [43], overapproximating unknown loop bounds to a smaller known bound [39], accelerating entire transition relations [8], using CHC transformations [6,35] and induction based techniques [9,10,11]. The portfolio solver VeriAbs [1] used in our experiment predominantly used the shrinking [39] technique to verify, which does not generate invariants. The tool also has induction-based techniques [9,10,11] that implicitly generate invariants, but are not given to the user. RAPID [26] translates the semantics of the input program into formulas in trace logic. Then the formulas are verified using a theorem prover. Though sound lemmas are used to translate loops, it currently does not support the extraction of invariants from the lemmas. Apart from the inability to generate explicit invariants, all of these techniques need preconditions to verify the programs.

Our technique works on CHCs, which has gained much attention in recent years for different verification and inference tasks [57,36,51,21,19,50,32,22]. Most of these techniques do not handle arrays, and when they do, do not generate maximal preconditions.

The core part of our algorithm uses abductive inference. Abduction has been used for programs without arrays to infer invariants [17,18], preconditions [27,16], and specifications [2,50]. The technique in [56] finds specification over uninterpreted functions by overcoming the limitation of integer abduction engines through a data-driven approach. In contrast, our technique extends the abduction itself for quantified formulas over arrays.

Recent works in specification synthesis uses artifacts like input-output examples, comments in the code, partial code snippets, and user-supplied constraints and languages to infer specifications [12,55,47]. In comparison, our work uses the entire program and postcondition expressed as a logical formula to find maximal preconditions.

10 Limitations and Future Work

The restriction on array access statements simplifies the conversion between array and integer terms in range abduction. However, this can be relaxed to support terms like $a[b[i]]$, $a[i+1]$ among others, by enhancing the select-substitution (recall Def 10).

The restriction on form of postconditions, inductive invariants and preconditions is required for effective range abduction and SMT checks. Our approach can easily support alternating quantifiers, if the structure of the postcondition is close to the inductive invariant.

For non-deterministic programs, Algorithm 4 will not terminate when a CTM has two satisfiable unrollings: π and $\overline{\pi}$ (refer Definition 8). Hence, the maximality check will be inconclusive. Nevertheless, Algorithm 1 can still generate preconditions (with inductive invariants) for such programs, often maximal ones as observed in our experiments. We extend our approach to non-deterministic CHCs in [48].

In the case of non-terminating programs, an initial state with non-terminating execution can be added to either $\boldsymbol{pre}$ or $\boldsymbol{cpre}$, as it will have inductive invariants for both. If added to the latter, the maximality check could wrongly conclude that $\boldsymbol{pre}$ is maximal when it's not. Therefore, relaxing this restriction affects the soundness of the maximality check. An interesting future direction for maximality checking would be to extend the work presented in [29] to incorporate array handling.

Data Availability and Artifact

The artifact accompanying the paper is publicly available at [52].

Acknowledgement

We would like to thank Pamina Georgiou for helping us with RAPID [26] tool.

References

1. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction and test generation. In: 2019

34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1138–1141. IEEE (2019)
2. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: POPL. pp. 789–801. ACM (2016)
3. Astorga, A., Madhusudan, P., Saha, S., Wang, S., Xie, T.: Learning stateful preconditions modulo a test generator. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 775–787 (2019)
4. Beyer, D.: Progress on software verification: Sv-comp 2022. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 375–402. Springer International Publishing, Cham (2022)
5. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (short papers). EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: International Static Analysis Symposium. pp. 105–125. Springer (2013)
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The essence of computation, pp. 85–108. Springer (2002)
8. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Computer Aided Verification. pp. 157–172. Springer Berlin Heidelberg (2009)
9. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: SAS. LNCS, vol. 10422, pp. 428–449. Springer (2017)
10. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 22–39. Springer (2020)
11. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: Inductive reasoning of array programs using difference invariants. In: CAV (2). Lecture Notes in Computer Science, vol. 12760, pp. 911–935. Springer (2021)
12. Chen, Y., Martins, R., Feng, Y.: Maximal multi-layer specification synthesis. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. pp. 602–612. ACM (2019). https://doi.org/10.1145/3338906.3338951, https://doi.org/10.1145/3338906.3338951
13. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 128–148. Springer (2013)
14. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 150–168. Springer (2011)
15. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
16. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: International Conference on Computer Aided Verification. pp. 684–689. Springer (2013)
17. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA. pp. 443–456. ACM (2013)

18. Echenim, M., Peltier, N., Sellami, Y.: Ilinva: Using abduction to generate loop invariants. In: FroCoS. LNCS, vol. 11715, pp. 77–93. Springer (2019)
19. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. PACMPL **2**(OOPSLA), 131:1–131:25 (2018)
20. Fedyukovich, G., Gurfinkel, A., Gupta, A.: Lazy but Effective Functional Synthesis. In: VMCAI. LNCS, vol. 11388, pp. 92–113. Springer (2019)
21. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving Constrained Horn Clauses Using Syntax and Data. In: FMCAD. pp. 170–178. IEEE (2018)
22. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified Invariants via Syntax-Guided Synthesis. In: CAV, Part I. LNCS, vol. 11561, pp. 259–277. Springer (2019)
23. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-Guided Termination Analysis. In: CAV, Part I. LNCS, vol. 10981, pp. 124–143. Springer (2018)
24. Flanagan, C., Leino, K.R.M.: Houdini: an Annotation Assistant for ESC/Java. In: FME. LNCS, vol. 2021, pp. 500–517. Springer (2001)
25. Gehr, T., Dimitrov, D., Vechev, M.: Learning commutativity specifications. In: International Conference on Computer Aided Verification. pp. 307–323. Springer (2015)
26. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: FMCAD. pp. 255–263. IEEE (2020)
27. Giacobazzi, R.: Abductive analysis of modular logic programs. In: ILPS. vol. 94, pp. 377–391 (1994)
28. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. pp. 405–416. ACM (2012)
29. Gu, Y., Tsukada, T., Unno, H.: Optimal chc solving via termination proofs. POPL p. conditionally accepted (2023)
30. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 235–246 (2008)
31. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. LNCS, vol. 9206, pp. 343–361. Springer (2015)
32. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: ATVA. LNCS, vol. 11138, pp. 248–266 (2018)
33. Henzinger, T.A., Hottelier, T., Kovács, L., Rybalchenko, A.: Aligators for arrays (tool paper). In: Logic for Programming, Artificial Intelligence, and Reasoning: 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings 17. pp. 348–356. Springer (2010)
34. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. ACM SIGPLAN Notices **39**(1), 232–244 (2004)
35. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: FMCAD. pp. 158–164. IEEE (2018)
36. Kafle, B., Gallagher, J.P., Ganty, P.: Solving non-linear Horn clauses using a linear Horn clause solver. In: HCVS. EPTCS, vol. 219, pp. 33–48 (2016)
37. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying Java programs. In: CAV, Part I. LNCS, vol. 9779, pp. 352–358. Springer (2016)
38. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: International Conference on Fundamental Approaches to Software Engineering. pp. 470–485. Springer (2009)

39. Kumar, S., Sanyal, A., Venkatesh, R., Shah, P.: Property checking array programs using loop shrinking. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 213–231. Springer International Publishing, Cham (2018)
40. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 267–281. Springer (2004)
41. Menguy, G., Bardin, S., Lazaar, N., Gotlieb, A.: Automated program analysis: Revisiting precondition inference through constraint acquisition. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2022), Vienna, Austria (2022)
42. Minsky, M.L.: Computation. Prentice-Hall Englewood Cliffs (1967)
43. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: Static Analysis. pp. 361–382. Springer Berlin Heidelberg (2016)
44. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
45. Moy, Y.: Sufficient preconditions for modular assertion checking. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 188–202. Springer (2008)
46. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features. ACM SIGPLAN Notices **51**(6), 42–56 (2016)
47. Park, K., D'Antoni, L., Reps, T.: Synthesizing specifications. arXiv preprint arXiv:2301.11117 (2023)
48. Prabhu, S., D'Souza, D., Chakraborty, S., Venkatesh, R., Fedyukovich, G.: Weakest precondition inference for non-deterministic linear array programs. 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (to appear) (2024)
49. Prabhu, S., Fedyukovich, G., D'Souza, D.: Maximal quantified precondition synthesis for linear array loops (extended version) (2024), `https://doi.org/10.6084/m9.figshare.25049996`
50. Prabhu, S., Fedyukovich, G., Madhukar, K., D'Souza, D.: Specification Synthesis with Constrained Horn Clauses. In: PLDI. pp. 1203–1217. ACM (2021)
51. Prabhu, S., Madhukar, K., Venkatesh, R.: Efficiently learning safety proofs from appearance as well as behaviours. In: SAS. LNCS, vol. 11002, pp. 326–343. Springer (2018)
52. Prabhu, Sumanth and Fedyukovich, Grigory and D'Souza, Deepak: Artifact for the paper titled "maximal quantified precondition synthesis for linear array loops" to appear in ESOP (2024), `https://doi.org/10.6084/m9.figshare.24945996`
53. Sankaranarayanan, S., Chaudhuri, S., Ivančić, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Proceedings of the 2008 international symposium on Software testing and analysis. pp. 295–306 (2008)
54. Seghir, M.N., Kroening, D.: Counterexample-guided precondition inference. In: European Symposium on Programming. pp. 451–471. Springer (2013)
55. Zhai, J., Shi, Y., Pan, M., Zhou, G., Liu, Y., Fang, C., Ma, S., Tan, L., Zhang, X.: C2S: translating natural language comments to formal program specifications. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. pp. 25–37. ACM (2020). https://doi.org/10.1145/3368089.3409716, `https://doi.org/10.1145/3368089.3409716`

56. Zhou, Z., Dickerson, R., Delaware, B., Jagannathan, S.: Data-driven abductive inference of library specifications. Proceedings of the ACM on Programming Languages **5**(OOPSLA) (2021)
57. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI. pp. 707–721. ACM (2018)

Verified Inlining and Specialisation for PureCake

Hrutvik Kanabar[1](✉), Kacper Korban[2], and Magnus O. Myreen[2]

[1] University of Kent, Canterbury, UK
hrk32@cantab.ac.uk
[2] Chalmers University of Technology, Gothenburg, Sweden
kacper.f.korban@gmail.com myreen@chalmers.se

Abstract. Inlining is a crucial optimisation when compiling functional programming languages. This paper describes how we have implemented and verified function inlining and loop specialisation for PureCake, a verified compiler for a Haskell-like (purely functional, lazy) programming language. A novel aspect of our formalisation is that we justify inlining by pushing and pulling `let`-bindings. All of our work has been mechanised in the HOL4 interactive theorem prover.

Keywords: verified compilation · function inlining · loop optimisation · functional programming · machine-checked proofs

1 Introduction

It can be tricky to generate high-quality code from lazy, purely functional programs for a number of reasons. One of these reasons is that functional programming encourages a brief declarative style that makes heavy use of shorthands (*e.g.*, for partially-applied functions) and higher-order functions [8]. Producing good code from such input requires a well-developed inliner, as noted [17] by the developers of the Glasgow Haskell Compiler (GHC):

> "One of the trickiest aspects of a compiler for a functional language is the handling of inlining. [...] Effective inlining is particularly crucial in getting good performance."

This paper is about implementing and verifying an inliner that can specialise loops for PureCake, an end-to-end verified compiler for a Haskell-like language [10].

The inliner by example. The following simple example demonstrates what our inliner does. Imagine that a programmer is to write a function that increments every element of a list of integers. The programmer should write:

```
suc_list = map (+1)
```

Here, the programmer has relied on the library function `map` below to perform the necessary list traversal.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

To generate high-quality code for `suc_list`, the compiler must *both* inline *and* specialise `map`. Our inliner takes the definition of `suc_list` above and produces the following code.

```
suc_list =
  let map' xs =
    case xs of
      []     -> []
      (y:ys) -> y + 1 : map' ys
  in map'
```

In particular, the inliner has combined the following code transformations:

– selective expansion of function definitions at call sites; and
– loop specialisation of recursive functions with known arguments (*e.g.*, argument `f` to `map` is always `(+1)` in `suc_list`).

Contributions. Our work adds verified inlining and loop specialisation to PureCake. Our inliner is capable of optimisations such as the one above. More specifically, we make the following contributions:

1. We define and prove sound a relation that encapsulates an envelope of semantics-preserving inlinings (§ 4). This relation is independent of the heuristics of any real implementation. It is proved sound using a novel formalisation of inlining as pushing/pulling of `let`-bindings.
2. We derive sound equational principles that allow us to lift out arguments which remain constant during recursion, such as `f` in `map` in the example above (§ 5). These principles are phrased such that they can be used in the relation above and have the effect of specialising loops.
3. We implement an inliner that can specialise loops and verify that its action preserves semantics, relying on the formalisations above (§ 6).
4. We integrate our inliner into the PureCake compiler and its verification (§ 7).

All of our work is mechanised using the HOL4 interactive theorem prover, and our development is open-source.[3] To the best of our knowledge, ours is the first verified inliner for a lazy functional programming language, and the first verified loop specialiser for any functional language.

2 The Inliner by Example

We begin with a high-level explanation of how our inliner works, before diving into verification details in later sections. We will show the transformations the

[3] https://github.com/cakeml/pure, see also our artifact hosted on Zenodo [9].

inliner performs step-by-step. As a running example, we use the code from the previous section with one modification: we lift `(+1)` to a separate function `add1` for clarity. The input code after this modification is as follows:

```
suc_list = map add1

add1 i = i + 1

map f [] = []
map f (x:xs) = f x : map f xs

main = ...
```

Our inliner is installed very early in the PureCake compiler, directly after parsing and *binding group analysis*. Binding group analysis processes the program above to the code below, breaking up the mutually recursive bindings into a nesting of `let`-expressions. Note that there is no dependency between `add1` and `map`, so their definitions could be reordered; for this example we put `add1` first.

```
18  let add1 i = i + 1 in
19  let map f l = case l of
20                  []     -> []
21                  (x:xs) -> f x : map f xs in
22  let suc_list = map add1 in
23  let main = ... in main
```

The inliner receives this program as input. As it traverses the program, it records known definitions that it may wish to inline later on. In particular, it maintains a mapping from names to their definitions, which starts off empty. Therefore, after processing line 18 (*i.e.*, the definition of `add1`), the mapping contains only the definition of `add1`, that is, `\i -> i + 1`.

The inliner then moves to line 19, the `let`-expression that defines `map`. The definition of `map` is recursive, so the inliner analyses it to determine whether any of its arguments remain constant over all recursive calls. In the case of `map`, it finds that the first argument, `f`, remains constant. This means that it can *loop specialise* `map` to produce the following equivalent definition.

```
let map f =
    let map' l = case l of
                   []     -> []
                   (x:xs) -> f x : map' xs
    in map'
in ...
```

Our inliner does not alter the definition of `map` in the program, but it does add this equivalent definition to its mapping of known definitions. We will very soon see why it is useful to pull out the constant argument `f`.

The inliner moves on to the definition of `suc_list` on line 22.

```
let suc_list = map add1 in ...
```

After pulling out the constant argument `f` above, the inliner considers `map` to be a single-argument function. Therefore, the application `map add1` here seems fully applied and the inliner will rewrite it. First, it transforms `map add1` into the following.

```
let suc_list =
    let f = add1 in
    let map' l = case l of
                   []     -> []
                   (x:xs) -> f x : map' xs
    in map'
in ...
```

Notice the use of a binding `let f = add1` to assign the constant argument `f` of `map`. Then, the inliner recurses into this expression, replacing `f` by `add1` in the second row of the pattern match:

```
                   (x:xs) -> add1 x : map' xs
```

The inliner recurses again into the modified subexpression `add1 x`, and realises that `add1` (which is mapped to `\i -> i + 1`) is fully applied. Therefore, it inlines `add1` too:

```
                   (x:xs) -> (let i = x in i + 1) : map' xs
```

Once again, the inliner recurses on the modified subexpression, turning the innermost `i` into `x`:

```
                   (x:xs) -> (let i = x in x + 1) : map' xs
```

The final code produced by the inliner is below. The definition of `suc_list` has been rewritten so extensively that it now resembles a copy of `map` which has been specialised to the `add1` function.

```
let add1 i = i + 1 in
let map f l = case l of
                []     -> []
                (x:xs) -> f x : map f xs in
let suc_list =
    let f = add1 in
    let map' l = case l of
                   []     -> []
                   (x:xs) -> (let i = x in x + 1) : map' xs
    in map'
in let main = ... in main
```

Some dead code remains, *e.g.*, `let f = add1` (line 46) and `let i = x` (line 49). We perform a simple dead code elimination pass immediately after the inliner to remove these.

Single-pass optimisation. Note that our inliner does not make multiple passes over input code, in contrast to the presentation above. It performs a single top-down pass over its input, calling itself recursively only on function applications or variables that it has successfully rewritten. The depth of this recursion is bounded by a simple user-configurable recursion limit.

3 Setting: PureCake

We implement and verify our inlining and specialisation optimisations as part of the verified compiler PureCake. In this section, we describe both the PureCake project at a high level, and the key aspects of its formalisation on which we rely.

What is PureCake? PureCake [10] is an end-to-end verified compiler for a Haskell-like language known as PureLang. Here, a "Haskell-like" language is one which: is purely functional with monadic effects; evaluates lazily; and has a syntax resembling that of Haskell. PureCake compiles PureLang to the CakeML language, which is call-by-value and ML-like, and has an end-to-end verified compiler [12,14]. CakeML targets machine code, so PureCake and CakeML can be composed to produce end-to-end guarantees for the compilation of PureLang to machine code [10, §6].

The PureCake compiler is designed to be realistic: it accepts a featureful input language and generates performant code. This makes it an ideal setting for verified inlining and specialisation optimisations. We add these to PureCake as PureLang-to-PureLang transformations.

Formalisation details. PureLang is formalised using two ASTs: *compiler* expressions and *semantic* expressions, denoted ce and e respectively [10, §3.2]. The compiler implementation uses compiler expressions, and their semantics is given by desugaring into semantic expressions (denoted desugar, of type $ce \to e$).

The call-by-name operational semantics of PureLang is defined over its simpler semantic expressions [10, §3.3]. This semantics admits an equational theory [10, §3.4] which is sound and complete with respect to contextual equivalence. Its equivalence relation, $e_1 \cong e_2$, is based on an untyped applicative bisimulation from Abramsky's lazy λ-calculus [1] and is proved congruent via Howe's method [7], *i.e.*, expressions composed of equivalent subexpressions are themselves equivalent.

PureCake's compiler passes are verified in two stages.

1. A binary syntactic relation is defined over semantic expressions ($e_1 \mathcal{R} e_2$). The relation is proved to imply $e_1 \cong e_2$, so e_1 and e_2 have identical observable behaviour in all contexts. Intuitively, the syntactic relation carves out an envelope of possible valid transformations, independent of the heuristics of any real implementation.
2. The implementation is then defined over compiler expressions, with concrete heuristics. It is verified to perform only those valid transformations expressed by the syntactic relation.

Composition of the two stages produces the overall proof that the action of the compiler implementation preserves semantics. A key benefit of this approach is that heuristics remain an implementation detail in stage 2, and can be changed without incurring the significant proof obligations of stage 1.

Approach and paper outline. We can now describe more precisely the steps we took to add inlining and loop specialisation to the PureCake compiler.

§ 4 *(stage 1)* We defined a relation which captures an envelope of valid inlining transformations, and proved that this relation preserves semantics.
§ 5 We formalised loop specialisation using PureLang's equational theory such that it can be used in the envelope mentioned above.
§ 6 *(stage 2)* We implemented the overall inlining and specialisation transformations over compiler expressions, verifying that they fit the envelopes.
§ 7 We integrated our inliner into the PureCake compiler pipeline and its top-level correctness result.
§ 8 We benchmarked the performance of the output of the inliner.

4 Inlining as a Relational Envelope

In this section, we define a relation which characterises all the inlinings that we wish to perform. We then prove that any code transformation contained within this relational envelope must preserve semantics.

4.1 Understanding the relation

We begin by describing the intuition behind our relation.

Inlining is not substitution. Inlining is a more complex transformation than substitution or β-conversion. If we were to view inlining as a special case of these, we would generate unsatisfactory code. In particular, consider the example below: inlining based on substitution must replace all three occurrences of `f` with its definition; inlining based on β-conversion would remove the `let`-binding.

```
let f i = 5 in f 1 : map f xs ++ map f ys
```

By contrast, a real inliner must be able to choose whether to inline a definition *per use of that definition*. In other words, the inliner should decide which usages of a given definition are rewritten on a case-by-case basis. For the example above, a real inliner should produce the code below. Note that it chooses to inline the function `f` only at the usage which fully applies it.

```
let f i = 5 in (\i -> 5) 1 : map f xs ++ map f ys
```

Of course, a real inliner would further transform `(\i -> 5) 1` into `5` (this is in fact a β-conversion). For clarity in this example, we do not show that step.

Inlining is a series of `let` transformations. The key intuition behind our inlining transformations is as follows. We push **let**-bindings into expressions as far as possible, rewrite the result, then pull the bindings out again. We illustrate this by example below, starting from the same initial code as above.

```
let f i = 5 in f 1 : map f xs ++ map f ys
```

We now push in the `let`-binding which defines `f` to produce a series of equivalent expressions. First, we push it in one step past the list constructor `(:)`:

```
(let f i = 5 in f 1) :
    (let f i = 5 in map f xs ++ map f ys)
```

Next, we push it in through the function application `f 1`:

```
(let f i = 5 in f) (let f i = 5 in 1) :
    (let f i = 5 in map f xs ++ map f ys)
```

Now, we choose to rewrite the use of `f` under the first `let f i = 5` to `\i -> 5`:

```
(let f i = 5 in (\i -> 5)) (let f i = 5 in 1) :
    (let f i = 5 in map f xs ++ map f ys)
```

Note that we have chosen *not* to perform any other rewrites of `f`, because other uses of `f` are not fully applied.

We can now reverse the pushing in of `let`-bindings, *i.e.*, we pull them out instead. The final result is as follows, where `f` is inlined exactly as we wanted:

```
let f i = 5 in (\i -> 5) 1 : map f xs ++ map f ys
```

Stacking `let` transformations. Above, our example shows how we can inline a *single* `let`-binding: we push it inwards, use it for rewriting, and pull it outwards back to its original position. We can generalise this straightforwardly to handle a *list* of `let`-bindings. This mimics the implementation of a real inliner, which must carry with it a collection of definitions it may wish to inline.

Consider the following example, in which an inliner attempts to rewrite the expression `g 3 + 7` and carries definitions `f i = 5`; `h i = 2`; `g i = f i + 1`.

```
let f i = 5 in
let h i = 2 in
let g i = f i + 1 in
  g 3 + 7
```

Just as with a single `let`-binding, we can push in the stack of `let`-bindings, rewrite, and pull them out again. This produces the following expression.

```
let f i = 5 in
let h i = 2 in
let g i = f i + 1 in
  (\i -> (\i -> 5) i + 1) 3 + 7
```

The only complication in generalising to a stack of `let`-bindings is that some definitions can depend on others. In the example above, the definition of `g` depends on `f`. This is why we model the bindings as a *list*: this preserves scoping correctly, ensuring we do not break any dependencies between definitions.

Note that this intuition of pushing in and pulling out of `let`-bindings applies only to the formalisation that justifies our inlining rewrites. The implementation of our inliner performs no such push/pull transformations: as one might expect, it merely carries around a simple (unordered) map of variable names to their definitions. This map represents exactly the set of definitions that the inliner may wish to use for rewriting at usage sites.

4.2 Defining a Semantics-Preserving Envelope

We now describe an inductive relation, $l \Vdash e_1 \leadsto e_2$, which characterises all of the inlining transformations that we perform. We prove that any transformation described by the relation lies within the equational theory of PureLang ($\cong$, § 3). Therefore, the relation describes only semantics-preserving transformations.

The relation $l \Vdash e_1 \leadsto e_2$ should be read as follows: expression e_1 can be transformed into expression e_2 under the definitions in the list l. Both e_1 and e_2 are PureLang semantic expressions, and l is a list of definitions. Each such definition is of the form $x \leftarrow e$, associating name x with semantic expression e. We will first describe the formal *meaning* of $l \Vdash e_1 \leadsto e_2$, which is best understood via its soundness theorem, Theorem 1. Then in following subsections, we describe key parts of the *definition* of $\leadsto$.

Theorem 1 relates derivations of $l \Vdash e_1 \leadsto e_2$ with $\cong$, PureLang's equational theory, assuming pre and $\mathsf{lets\_ok}$. The definitions of pre and $\mathsf{lets\_ok}$ are shown in Figure 1—they enforce distinct variable names between both the expression e_1 and each of the definitions in l to avoid inadvertent clashes or capture.

$$\mathsf{vars\_of}\ l \overset{\mathsf{def}}{=} \bigcup \left\{ \{x\} \cup \mathsf{freevars}\ e \;\middle|\; \mathsf{mem}\ (x \leftarrow e)\ l \right\}$$

$$\mathsf{pre}\ l\ e \overset{\mathsf{def}}{=} \mathsf{barendregt}\ e\ \wedge\ \mathsf{boundvars}\ e \# \mathsf{vars\_of}\ l$$

$$\begin{aligned} &\mathsf{lets\_ok}\ [] \overset{\mathsf{def}}{=} \mathsf{T} \\ &\mathsf{lets\_ok}\ ((x \leftarrow e) :: l) \overset{\mathsf{def}}{=} \\ &\qquad x \notin \mathsf{freevars}\ e\ \wedge\ \big(\{x\} \cup \mathsf{freevars}\ e\big) \# \{x \mid \exists e.\ \mathsf{mem}\ (x \leftarrow e)\ l\}\ \wedge\ \mathsf{lets\_ok}\ l \end{aligned}$$

Fig. 1. The definition of pre and lets_ok. Here, the # predicate returns true only for disjoint sets: $s_1 \# s_2 \overset{\mathsf{def}}{=} (s_1 \cap s_2 = \varnothing)$.

Theorem 1. Soundness of $l \Vdash e_1 \rightsquigarrow e_2$.

$$\vdash\ l \Vdash e_1 \rightsquigarrow e_2\ \wedge\ \mathsf{pre}\ l\ e_1\ \wedge\ \mathsf{lets\_ok}\ l\ \Rightarrow\ \mathbf{lets}\, l\, e_1 \cong \mathbf{lets}\, l\, e_2$$

where $\quad \mathbf{lets}\,[]\ e \overset{\mathsf{def}}{=} e \quad$ and $\quad \mathbf{lets}\ ((x \leftarrow e') :: l)\ e \overset{\mathsf{def}}{=} \mathbf{let}\, x = e'\ \mathbf{in}\ (\mathbf{lets}\, l\ e)$

In particular, expressions e_1 and e_2 related in the context of definitions l produce equal expressions (according to $\cong$) under the stack of **let**-bindings corresponding to l. The latter correspondence is encapsulated by the definition of **lets**, which nests **let**-bindings. This theorem is proved by induction over the derivation of $l \Vdash e_1 \rightsquigarrow e_2$. In upcoming subsections, we will examine key rules of $\rightsquigarrow$ and their cases in this inductive proof.

When the inliner is first invoked, it is passed an entire PureLang program and has no knowledge of any definitions. In other words, its mapping of variable names to known definitions is empty, corresponding to the list l being empty ($[]$). In this case, we can simplify Theorem 1 by instantiating $l \mapsto []$, and unfolding the definitions of pre l and lets_ok l. This produces the following theorem:

Theorem 2. Soundness of $[] \Vdash e_1 \rightsquigarrow e_2$.

$$\vdash\ [] \Vdash e_1 \rightsquigarrow e_2\ \wedge\ \mathsf{barendregt}\ e_1\ \wedge\ \mathsf{closed}\ e_1\ \Rightarrow\ e_1 \cong e_2$$

We can read this as follows: if we can transform some closed e_1 which satisfies barendregt to some e_2 according to $\rightsquigarrow$, then e_1 and e_2 are equivalent. The barendregt predicate restricts the variable naming convention within e_1 to avoid problems with variable capture, because PureLang has explicit names. In particular, barendregt is the well known Barendregt variable convention that enforces unique free/bound variable names across an entire program [3].

The precise definition of barendregt is not necessary here. Suffice it to say that in order to discharge this assumption, our inliner implementation will rely on a freshening pass. This pass α-renames programs such that they obey the Barendregt variable convention, and therefore satisfy barendregt.

Reflexivity. We must allow the inliner to choose whether to rewrite a usage site on a case-by-case basis (§ 4.1). Therefore, the inliner must be allowed *not* to inline, *i.e.*, it must be able to leave an expression unchanged. Therefore the $\rightsquigarrow$ relation has a reflexivity rule:

$$\frac{}{l \Vdash e \rightsquigarrow e}\ \textsc{refl}$$

The REFL case of the proof of Theorem 1 boils down to showing the equation $\mathbf{lets}\,l\,e \cong \mathbf{lets}\,l\,e$, which is trivial due to reflexivity of $\cong$.

Inlining. The simplest rule for inlining uses a definition found in the list l (where mem denotes list membership) to rewrite a variable:

$$\frac{\mathsf{mem}\ (x \leftarrow e)\ l}{l \Vdash \mathbf{var}\,x \rightsquigarrow e}\ \textsc{inline}$$

In particular, if l associates name x with definition e, then the variable $\mathbf{var}\,x$ can be replaced by expression e. The INLINE case of Theorem 1 requires establishing:

$$\vdash\ \mathsf{mem}\ (x \leftarrow e)\ l\ \wedge\ \mathsf{lets\_ok}\ l\ \wedge\ \mathsf{pre}\ l\ (\mathbf{var}\,x) \Rightarrow \mathbf{lets}\,l\ (\mathbf{var}\,x) \cong \mathbf{lets}\,l\,e$$

Proof outline. We first derive a lemma that allows us to duplicate a **let**-binding from l, assuming lets_ok (defined in Figure 1 such that it enables this lemma):

$$\vdash\ \mathsf{mem}\ (x \leftarrow e)\ l \wedge \mathsf{lets\_ok}\ l \Rightarrow \mathbf{lets}\,l\,e' \cong \mathbf{lets}\,l\ (\mathbf{let}\,x = e\ \mathbf{in}\ e') \quad \textsc{Let-dup}$$

Equipped with the LET-DUP lemma, we proceed as follows:

$$\begin{aligned} \mathbf{lets}\,l\ (\mathbf{var}\,x) &\cong \mathbf{lets}\,l\ (\mathbf{let}\,x = e\ \mathbf{in}\ \mathbf{var}\,x) && (\textsc{Let-dup}) \\ &\cong \mathbf{lets}\,l\,e && (\text{trivial}) \end{aligned}$$

□

Let. We can now inline known definitions, but we must be able to learn those definitions in the first place. The rule LET allows us to add a **let**-bound definition to the stack l, using the append operator ($+\!\!+$).

$$\frac{l \Vdash e_1 \rightsquigarrow e_1' \qquad l +\!\!+ (x \leftarrow e_1) \Vdash e_2 \rightsquigarrow e_2'}{l \Vdash (\mathbf{let}\,x = e_1\ \mathbf{in}\ e_2) \rightsquigarrow (\mathbf{let}\,x = e_1'\ \mathbf{in}\ e_2')}\ \textsc{Let}$$

Proof outline. LET case of Theorem 1.

$$\begin{aligned} & \mathbf{lets}\,l\ (\mathbf{let}\,x = e_1\ \mathbf{in}\ e_2) \\ \cong\ & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_1))\ e_2 && (\text{definition of } \mathbf{lets}) \\ \cong\ & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_1))\ e_2' && (\textsc{ih} \text{ for } e_2) \\ \cong\ & \mathbf{lets}\,l\ (\mathbf{let}\,x = e_1\ \mathbf{in}\ e_2') && (\text{definition of } \mathbf{lets}) \\ \cong\ & \mathbf{let}\,x = (\mathbf{lets}\,l\,e_1)\ \mathbf{in}\ (\mathbf{lets}\,l\,e_2') && (\text{push in } \mathbf{lets}) \\ \cong\ & \mathbf{let}\,x = (\mathbf{lets}\,l\,e_1')\ \mathbf{in}\ (\mathbf{lets}\,l\,e_2') && (\textsc{ih} \text{ for } e_1) \\ \cong\ & \mathbf{lets}\,l\ (\mathbf{let}\,x = e_1'\ \mathbf{in}\ e_2') && (\text{pull out } \mathbf{lets}) \end{aligned}$$

□

Above, we can push and pull **lets** through **let** because the precondition pre enforces sufficiently distinct variable names.

Note that this rule records the unmodified expression e_1 in the stack of known definitions l. It could instead use the $\rightsquigarrow$-transformed expression e_1'. The proof strategy with this modification is essentially unchanged, except we must reverse our applications of the inductive hypotheses.

Congruences. We must be able to apply $\rightsquigarrow$ within subexpressions. Therefore, we have several *congruence* rules, such as the following:

$$\frac{l \Vdash e_1 \rightsquigarrow e_1' \qquad l \Vdash e_2 \rightsquigarrow e_2'}{l \Vdash (e_1 \cdot e_2) \rightsquigarrow (e_1' \cdot e_2')}\ \textsc{App-cong} \qquad \frac{l \Vdash e \rightsquigarrow e'}{l \Vdash (\lambda x.\ e) \rightsquigarrow (\lambda x.\ e')}\ \textsc{Lam-cong}$$

$$\frac{\forall i.\ l \Vdash e_i \rightsquigarrow e_i' \qquad l \Vdash e \rightsquigarrow e'}{l \Vdash (\textbf{letrec}\ \overline{x_n = e_n}\ \textbf{in}\ e) \rightsquigarrow \left(\textbf{letrec}\ \overline{x_n = e_n'}\ \textbf{in}\ e'\right)}\ \textsc{Letrec-cong}$$

Each such case in Theorem 1 requires showing that we can push/pull **lets** into/out of subexpressions. Once again, the precondition pre permits this by enforcing sufficiently distinct variable names. The remainder of the proof follows from congruence of $\cong$.

Simplification. The following rule allows $\rightsquigarrow$ to carry out any transformation that preserves $\cong$:

$$\frac{l \Vdash e_1 \rightsquigarrow e_2 \qquad e_2 \cong e_2'}{l \Vdash e_1 \rightsquigarrow e_2'}\ \textsc{simp}$$

The SIMP case in Theorem 1 is a direct consequence of the transitivity of $\cong$.

This rule permits the inliner to modify (and in particular, simplify) generated expressions during its operation. There are two important uses of this ability:

– Turning fully applied λ-abstractions into a stack of **let**-bindings. This allows recursive applications of inlining (see rule TRANS below).

$$(\lambda x_1.\ \lambda x_2.\ \ldots\ \lambda x_n.\ e) \cdot e_1 \cdot e_2 \cdot \ldots \cdot e_n \cong \textbf{lets}\ (x_1 \leftarrow e_1 :: x_2 \leftarrow e_2 :: \ldots :: x_n \leftarrow e_n)\ e \qquad (1)$$

– Freshening names of bound variables (*i.e.*, α-renaming). This happens directly before application of the rule TRANS below.

Transitivity. To permit recursion into recently inlined expressions, $\rightsquigarrow$ has a transitivity rule:

$$\frac{l \Vdash e_1 \rightsquigarrow e_2 \qquad l \Vdash e_2 \rightsquigarrow e_3 \qquad \mathsf{pre}\ l\ e_2}{l \Vdash e_1 \rightsquigarrow e_3}\ \textsc{trans}$$

In particular, e_1 can be transformed to e_3 if there is some intervening e_2 which can act as a stepping stone.

Unusually, we require precondition pre to hold of intermediate expression e_2. This is demanded by the proof of Theorem 1, in which we can only instantiate inductive hypotheses if we first establish pre. Unfortunately, $l \Vdash e_1 \rightsquigarrow e_2$ and $\mathsf{pre}\ l\ e_1$ are not enough to derive $\mathsf{pre}\ l\ e_2$. Fortunately, we can freshen bound variable names (*i.e.*, α-rename) sufficiently to establish pre, and justify this freshening using rule SIMP above.

Specialisation. The $\rightsquigarrow$ relation must be able to support loop specialisation, as described for the `map` function in § 2. Therefore, it has a rule SPEC which permits conversion of a **letrec** into a **let**, as long as there is a proof that the conversion preserves $\cong$.

$$\frac{\begin{array}{cc} l \Vdash e_1 \rightsquigarrow e_1' & (\forall e.\ \mathbf{letrec}\, x = e_1\ \mathbf{in}\ e \cong \mathbf{let}\, x = e_2\ \mathbf{in}\ e) \\ l +\!\!+ (x \leftarrow e_2) \Vdash e_3 \rightsquigarrow e_3' \quad \mathsf{disjoint\_names}\ e_2\ e_3 & x \notin \mathsf{freevars}\ e_2 \end{array}}{l \Vdash \mathbf{letrec}\, x = e_1\ \mathbf{in}\ e_3 \rightsquigarrow \mathbf{letrec}\, x = e_1'\ \mathbf{in}\ e_3'}\ \text{SPEC}$$

That is, if we can $\cong$-convert some **letrec** $x = e_1$ to some **let** $x = e_2$, then we can append $x \leftarrow e_2$ to the stack of known definitions when processing **letrec** body e_3. Again, we require restrictions on variable naming: the variables bound in e_2 and e_3 must be disjoint, and the bound variable x must not appear free in e_2.

Proof outline. SPEC case of Theorem 1.

$$\begin{array}{lll}
& \mathbf{lets}\, l\ (\mathbf{letrec}\, x = e_1\ \mathbf{in}\ e_3) & \\
\cong & \mathbf{lets}\, l\ (\mathbf{let}\, x = e_2\ \mathbf{in}\ e_3) & \text{(assumption of rule)} \\
\cong & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_2))\ e_3 & \text{(definition of } \mathbf{lets}) \\
\cong & \mathbf{lets}\ (l +\!\!+ (x \leftarrow e_2))\ e_3' & (\text{IH for } e_3) \\
\cong & \mathbf{lets}\, l\ (\mathbf{let}\, x = e_2\ \mathbf{in}\ e_3') & \text{(definition of } \mathbf{lets}) \\
\cong & \mathbf{lets}\, l\ (\mathbf{letrec}\, x = e_1\ \mathbf{in}\ e_3') & \text{(ass. of rule, symmetry of } \cong) \\
\cong & \mathbf{letrec}\, x = (\mathbf{lets}\, l\, e_1)\ \mathbf{in}\ \mathbf{lets}\, l\, e_3' & \text{(push } \mathbf{lets}) \\
\cong & \mathbf{letrec}\, x = (\mathbf{lets}\, l\, e_1')\ \mathbf{in}\ \mathbf{lets}\, l\, e_3' & (\text{IH for } e_1) \\
\cong & \mathbf{lets}\, l\ (\mathbf{letrec}\, x = e_1'\ \mathbf{in}\ e_3') & \text{(pull out } \mathbf{lets})
\end{array}$$

□

5 Specialisation of Recursive Bindings

Our example in § 2 showed that our inliner can specialise applications of recursive functions such as `map` to known arguments such as `add1`. This is possible whenever constant arguments such as `f` can be pulled out of the recursion. That is, whenever we can transform recursive functions like `map` (left) into equivalent code which makes the constant argument explicit using `map'` (right):

```
let map f l =
  case l of
    []     -> []
    (x:xs) -> f x : map f xs
```

```
let map f = let map' l =
              case l of
                []     -> []
                (x:xs) -> f x : map' xs
            in map'
```

In this section, we describe how we prove correctness of such transformations. Critically, our proofs can be used in the SPEC rule of $\rightsquigarrow$ from the previous section.

5.1 Understanding Specialisation

Like $\rightsquigarrow$, our specialisation transformation is justified using equational reasoning. We illustrate the equational steps below, again noting that the implementation is much more direct. We use the `map` example of § 1, eliding parts not relevant to specialisation. The input is therefore as follows:

```
let map f l = ... f x ... map f xs ...
```

We first make a local copy of the recursive definition `map`, named `map'`:

```
let map = let map' f l = ... f x ... map' f xs ...
          in map'
```

We then η-expand the final usage of the copy `map'`:

```
let map = let map' f l = ... f x ... map' f xs ...
          in \f l -> map' f l
```

Next, we pull out the new λ-abstractions to the top-level:

```
let map f l = let map' f l = ... f x ... map' f xs ...
              in map' f l
```

We then α-rename the constant argument in the copy (here, `f` becomes `g`):

```
let map f l = let map' g l = ... g x ... map' g xs ...
              in map' f l
```

The first major step (*transform 1*) replaces the constant argument `g` with the known value to which the function `map'` is always applied, `f`:

```
let map f l = let map' g l = ... f x ... map' f xs ...
              in map' f l
```

The second major step (*transform 2*) deletes the now unused argument `g`. It removes the argument from *both* the definition of `map'` *and* all calls to `map'`:

```
let map f l = let map' l = ... f x ... map' xs ...
              in map' l
```

We push back in some of the top-level λ-abstractions, in this case just `l`:

```
let map f = let map' l = ... f x ... map' xs ...
            in \l -> map' l
```

Finally, η-contraction removes the λ-abstraction over `l`:

```
let map f = let map' l = ... f x ... map' xs ...
            in map'
```

Most of the steps are straightforwardly justified in PureLang's equational theory. However, the steps marked *transform 1* and *transform 2* are more involved. We discuss these below.

5.2 Key Lemmas for Specialisation

Both *transform 1* and *transform 2* require a substitution-like traversal of the entire subexpression under consideration. It is not clear how to justify these traversals using simple equational reasoning in PureLang's theory. Therefore, we resort to more cumbersome simulation proofs to establish $\cong$ by appealing to its definition in terms of PureLang's operational semantics.

For *transform 1*, we prove a theorem of the following form. Here call_with_arg holds only if every application of f in e is applied to **var** y after n arguments, and the names f and y are never rebound within e.

$$\begin{aligned}
&\vdash \mathsf{call\_with\_arg}\ f\ \overline{x_n}\ y\ e\ \wedge\ \ldots\\
&\quad \Rightarrow\ \mathbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ e)\ \ \mathbf{in}\ \ ((\mathbf{var}\, f)\cdot \overline{e_{1\,n}}\cdot(\mathbf{var}\, w)\cdot \overline{e_{2\,m}})\\
&\qquad\quad \cong\ \mathbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ e[^{\mathbf{var}\ w}/_{y}])\ \ \mathbf{in}\ \ ((\mathbf{var}\, f)\cdot \overline{e_{1\,n}}\cdot(\mathbf{var}\, w)\cdot \overline{e_{2\,m}})
\end{aligned}$$

Though the variable w is free in the theorem above, it is a closed constant expression in most parts of the proof, which simplifies the derivation of this theorem. This is because $\cong$ is defined over open terms in terms of closing substitution and a relation over closed terms. The proof of this theorem is a large simulation based on the semantics of PureLang.

For *transform 2*, we prove a theorem with a similar shape. This time, remove_call_arg is an inductive relation that ensures y never appears in e_1 and relates e_1 to a second expression e_2 in which the relevant argument has been removed from each application of f.

$$\begin{aligned}
&\vdash \mathsf{remove\_call\_arg}\ f\ \overline{x_n}\ y\ \overline{z_m}\ e_1\ e_2\ \wedge\ \ldots\\
&\quad \Rightarrow\ \mathbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda y.\ \lambda\, \overline{z_m}\,.\ e_1)\ \ \mathbf{in}\ \ ((\mathbf{var}\, f)\cdot \overline{e_{3\,n}}\cdot(\mathbf{var}\, y)\cdot \overline{e_{4\,m}})\\
&\qquad\quad \cong\ \mathbf{letrec}\, f = (\lambda\, \overline{x_n}\,.\ \lambda\, \overline{z_m}\,.\ e_2)\ \ \mathbf{in}\ \ ((\mathbf{var}\, f)\cdot \overline{e_{3\,n}}\cdot \overline{e_{4\,m}})
\end{aligned}$$

We prove this theorem by a large simulation too. The simulation strategy is necessary because **letrec** causes (potentially non-terminating) recursion.

6 Implementing a Correct Inliner

In this section, we describe the implementation of our inliner and the proof that its action lies within the $\rightsquigarrow$ relation described in § 4. We also touch on three other transformations mentioned previously: specialisation, freshening of bound variables, and dead code elimination. Our inliner relies on all three.

6.1 Preliminaries

We implement our inliner within a state monad with the following type:

$$\alpha\ \mathsf{M} \overset{\mathsf{def}}{=} \mathsf{name\ set} \to (\alpha,\ \mathsf{name\ set})$$

Here, $\mathsf{name\ set}$ is a set of variable names; we will see its usage shortly. This monad has standard return/bind operators, and we will use Haskell-style do-notation to show definitions written within the monad.

The inliner itself has the following signature:

$$\mathsf{inline}\ :\ (h : \mathsf{heuristic}) \to (k : \mathsf{num}) \to (m : (\mathsf{name} \mapsto ce)) \to ce \to ce\ \mathsf{M}$$

In other words, the inliner transforms compiler expressions to compiler expressions within the state monad, requiring several other inputs:

- An *unordered* mapping m from names to expressions. This is the "memory" of the inliner: the set of known definitions which it can use for rewriting.
- Heuristic h decides whether to "remember" a definition for future inlining. It accepts an expression ce and returns a boolean: if true, the definition should be remembered.
- Natural number k is the recursion limit for the inliner, used to bound its recursion into rewritten expressions.
- The $\mathsf{name\ set}$ parameter hidden within the monad keeps track of all variable names (whether bound or free) in input expression ce. It is used to ensure that sufficiently fresh variable names are chosen when freshening the names of bound variables.

6.2 Inliner implementation

The inliner traverses compiler expressions top-down. During the traversal, it performs two key operations: rewriting a variable to a known definition from memory, and adding a new definition to memory.

Rewriting a variable. There are two kinds of expressions in which the inliner will attempt to rewrite a variable. The first is a lone variable (of the form $\mathbf{var}\,x$), and the second is an application of a variable to some arguments (of the form $(\mathbf{var}\,x) \cdot \ldots$). The latter case is used to inline fully applied functions only.

In the lone variable case, the inliner is defined as follows:

$$\mathsf{inline}_h^k\ m\ (\mathbf{var}\,x) \overset{\text{def}}{=} \begin{cases} \mathsf{return}\ (\mathbf{var}\,x) & x \notin \mathsf{domain}\ m\ \lor\ k = 0\ \lor \\ & \quad m(x) = \lambda y.\ \ldots \\ \mathsf{inline}_h^{k-1}\ m\ ce & m(x) = ce \end{cases}$$

That is, on encountering a free variable x the inliner does one of the following:

– *Leaves the variable unchanged* if the definition of x is unknown, or the recursion limit has been reached, or the definition of x is known to be a λ-abstraction. The last case may seem unusual, but note we do not rewrite variables to λ-abstractions unless the result will be fully applied. This is handled in the application case below.
– *Rewrites the variable* by inserting the expression ce found in memory, and then recurses into ce with a decremented recursion limit.

In the application case, the inliner is defined as follows:

```
$\mathsf{inline}_h^k\ m\ ((\mathbf{var}\,x) \cdot ce_1 \cdot \ldots \cdot ce_n) \overset{\text{def}}{=}$ do
  $[ce'_1,\ \ldots,\ ce'_n] \leftarrow \mathsf{mapM}\ (\mathsf{inline}_h^k\ m)\ [ce_1,\ \ldots,\ ce_n]$;
  if $x \notin \mathsf{domain}\ m\ \lor\ k = 0$ then return $((\mathbf{var}\,x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$ else do
    $ce \leftarrow \mathsf{freshen}\ (m(x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$;                (2)
    case convert_to_lets $ce$ of
    | None $\rightarrow$ return $((\mathbf{var}\,x) \cdot ce'_1 \cdot \ldots \cdot ce'_n)$
    | Some $ce'$ $\rightarrow$ $\mathsf{inline}_h^{k-1}\ m\ ce'$
```

That is, on encountering a free variable x applied to n arguments the inliner does the following:

1. Recurses into the arguments to produce n new arguments.
2. Searches for variable x in memory and checks the recursion limit. If x is not found or the recursion limit has been reached, the inliner returns variable x applied to the n *new* arguments.
3. Rewrites x using its definition from memory, $m(x)$.
4. Freshens the resulting application of $m(x)$ to the n new arguments.
5. Attempts to convert the freshened application to a series of **let**-bindings. This is precisely the conversion shown in eq. (1) (pg. 11). Note that the conversion fails (returns None) if $m(x)$ is not fully applied, in which case the inliner bails out of inlining the definition of x.
6. Recurses into the newly produced series of **let**-bindings with a decremented recursion limit.

The conversion into **let**-bindings is critical: it allows the inliner to learn the definitions of the applied arguments $ce'_1,\ \ldots,\ ce'_n$ for future inlining within the function body of $m(x)$. Note that we only decrement the recursion limit when the size of the input expression may not have strictly decreased. This happens only when performing non-structural recursions, which only occur when we recurse into a definition rewritten from memory.

Remembering a new definition. The inliner can remember **let**- or **letrec**-bound expressions.
In the **let** case, it is defined as follows:

```
$\mathsf{inline}^k_h\ m\ (\mathbf{let}\, x = ce_1\ \mathbf{in}\ ce_2) \overset{\mathsf{def}}{=} \mathsf{do}$
  $ce_1' \leftarrow \mathsf{inline}^k_h\ m\ ce_1;$
  $\mathsf{let}\ m' = \mathsf{remember}_h\ m\ (x \leftarrow ce_1)\,;$
  $ce_2' \leftarrow \mathsf{inline}^k_h\ m'\ ce_2;$
  $\mathsf{return}\ (\mathbf{let}\, x = ce_1'\ \mathbf{in}\ ce_2')$
```

$$\mathsf{remember}_h\ m\ (x \leftarrow ce) \overset{\mathsf{def}}{=} \mathsf{if}\ \mathsf{cheap}\ ce \wedge h\ ce\ \mathsf{then}\ m[x \mapsto ce]\ \mathsf{else}\ m$$

That is, the inliner recurses into ce_1 (without decrementing the recursion limit), before memorising the definition $x \leftarrow ce_1$ and recursing into ce_2 with the augmented memory. The function remember records the definition only when two conditions are satisfied: the definition is cheap, and heuristic h returns true.

As the name suggests, cheap is a predicate that determines whether a definition is cheap to compute, and so will not slow the program down or cause loss of value sharing when inlined. The definition of cheap is as follows:

$$\mathsf{cheap}\ (\mathbf{var}\, x) = \mathsf{cheap}\ (\lambda x.\ e) = \mathsf{cheap}\ (op[\,]) \overset{\mathsf{def}}{=} \mathsf{T} \qquad \mathsf{cheap}\ \_ \overset{\mathsf{def}}{=} \mathsf{F}$$

In the **letrec** case, the inliner must also perform specialisation. Its action is defined as follows:

```
$\mathsf{inline}^k_h\ m\ (\mathbf{letrec}\, x = ce_1\ \mathbf{in}\ ce_2) \overset{\mathsf{def}}{=} \mathsf{do}$
  $ce_1' \leftarrow \mathsf{inline}^k_h\ m\ ce_1;$
  $\mathsf{let}\ m' = \mathsf{remember\_rec}_h\ m\ (x \leftarrow ce_1)\,;$
  $ce_2' \leftarrow \mathsf{inline}^k_h\ m'\ ce_2;$
  $\mathsf{return}\ (\mathbf{letrec}\, x = ce_1'\ \mathbf{in}\ ce_2')$
```

```
$\mathsf{remember\_rec}_h\ m\ (x \leftarrow ce) \overset{\mathsf{def}}{=}$
  $\mathsf{if}\ \neg\, \mathsf{can\_specialise}\ (x \leftarrow ce) \vee \neg\, \mathsf{h}\ ce\ \mathsf{then}\ m\ \mathsf{else}$
    $\mathsf{let}\ ([w_1^{a_1}\ \ldots\ w_n^{a_n}],\ \lambda\, \overline{y_m}\,.\ ce') = \mathsf{extract\_const\_args}\ (x \leftarrow ce)$
    $\mathsf{in}\ [x \mapsto \mathsf{specialise}\, x\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ (\lambda\, \overline{y_m}\,.\ ce')]$
```

This mirrors the **let** case almost exactly. The key difference is the use of remember_rec instead of remember: this does not check cheap, but does attempt specialisation (and bails out if it fails). We examine specialisation in the upcoming subsection.

Heuristics. So far, we have only implemented one heuristic based on expression size: the inliner only remembers definitions that are smaller than a user-configurable bound. Our implementation can accept any heuristic function as an input, making it straightforward to support new kinds of heuristic.

Implementing specialisation. Above, $\mathsf{specialise}$ transforms a **letrec**-binding into a **let**-binding before adding it to memory. We rely on two helper functions: $\mathsf{can\_specialise}$ and $\mathsf{extract\_const\_args}$.

The test $\mathsf{can\_specialise}$ simply checks if we are able to specialise a recursive body. The body must be a λ-abstraction with some constant arguments. Then, $\mathsf{extract\_const\_args}$ will extract these constant arguments. It accepts a definition $x \leftarrow ce$, where we know ce is a λ-abstraction of the form $\lambda\,\overline{x_n}\,.\,ce$. It splits the formal parameters $\overline{x_n}$ into $x_1 \ldots x_m$ and $x_{m+1} \ldots x_n$, where m is the minimum number of arguments that x is invoked with recursively in body ce. It further annotates the $x_1 \ldots x_m$ with annotations $a_1 \ldots a_m$, which describe whether the arguments remain constant for each recursive call. In the implementation of inline above, this has produced the annotated variables $w_i^{a_i}$ and left the remainder of the λ-abstraction untouched ($\lambda\,\overline{y_m}\,.\,ce'$).

Then, $\mathsf{specialise}$ is defined as follows.

$$\begin{aligned}
&\mathsf{specialise}\ f\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ ce \overset{\mathsf{def}}{=} \\
&\quad \mathsf{let}\ (\overline{x_n}, ce') = \mathsf{specialise\_each}\ x\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ ce\ \mathsf{in} \\
&\quad \mathsf{let}\ (\overline{y_i}, \overline{z_j}) = \mathsf{drop\_common\_suffix}\ [w_1^{a_1}\ \ldots\ w_n^{a_n}]\ \overline{x_n}\ \mathsf{in} \\
&\quad \lambda\,\overline{y_i}\,.\,\mathbf{letrec}\,f = (\lambda\,\overline{x_n}\,.\,ce')\ \ \mathbf{in}\ \ (\mathbf{var}\,f)\cdot(\mathbf{var}\,z_1)\cdot\ldots\cdot\ (\mathbf{var}\,z_j)
\end{aligned}$$

That is, it processes each annotated variable in turn, updating their call sites in body ce (*i.e.*, performing *transform 1* and *transform 2* from § 5 simultaneously using $\mathsf{specialise\_each}$), producing a new set of formal parameters $\overline{x_n}$. It determines which of these can be η-contracted (the final step in § 5) with a call to $\mathsf{drop\_common\_suffix}$, and then returns the new **letrec** which accepts constant arguments $\overline{y_i}$ at the top-level, and has η-contracted constant arguments $\overline{z_j}$ applied directly already.

Freshening and Dead-Let Elimination. Our inliner assumes that its input expression has a variable naming convention which is sufficient to prevent it from accidentally capturing variables during operation. Therefore, we only give the inliner expressions which obey the Barendregt variable convention, which asserts unique bound variable names and disjoint bound/free names [3]. This is achieved by freshening (α-renaming) bound variables directly before inlining, and further freshening before recursing into subexpressions taken from the inliner's memory. For example, the inliner invokes $\mathsf{freshen}$ in eq. (2) (pg. 16) above. This is precisely why the inliner carries around a $\mathsf{name\ set}$ in its state monad: this set contains all variable names (whether bound or free) of the input expression. Freshening avoids names in this set when inventing fresh names, and returns an updated set each time it runs.

The output of the inliner also contains various unused **let**-bindings. We showed such bindings in the example of § 1 (namely, `f` and `i`). To remove such bindings, we run a dead-**let** elimination pass directly after the inliner.

Including these two auxiliary passes, the top-level definition of the inliner is as follows:

$$\begin{aligned}
&\mathsf{inliner}_h^k\ ce \overset{\text{def}}{=} \\
&\quad \mathsf{let}\ (ce',\ names) = \mathsf{freshen}\ ce\ (\mathsf{boundvars}\ ce)\ \mathsf{in} \\
&\quad \mathsf{let}\ (ce_i,\ \_) = \mathsf{inline}_h^k\ \varnothing\ ce'\ names\ \mathsf{in} \\
&\quad\ \ \mathsf{dead\_let}\ ce_i
\end{aligned} \tag{3}$$

That is, the inliner freshens names, inlines definitions top-down starting with an empty ($\varnothing$) memory, then removes dead **let**s. Note that the top-level definition expects to receive only closed expressions, which is why it only passes bound variables (boundvars) to freshen. This respects our invariant that the name set contains all bound and free variable names, as there are no free variables.

6.3 Inliner correctness

In this section, we prove that the inliner implementation is correct. In the context of PureCake's proof strategy as described in § 3:

– *(stage 1)* Theorem 2 above (pg. 9) proved that $\rightsquigarrow$ preserves semantics.
– *(stage 2)* Theorem 3 below will prove that any transformation performed by the inliner lies within the $\rightsquigarrow$ relation of § 4.

We then compose these results to produce our final soundness theorem: the output expression of the inliner is equivalent to its corresponding input.

Theorem 3. inline satisfies $\rightsquigarrow$.

$$\begin{aligned}
&\vdash \mathsf{inline}_h^k\ m\ ce\ ns = (ce',\ ns') \ \land\ \mathsf{memory\_rel}_{ns}\ l\ m\ \land \\
&\quad \mathsf{barendregt}\ \ (\mathsf{desugar}\ ce)\ \land\ \mathsf{boundvars}\ ce \ \#\ \mathsf{domain}\ m\ \land \\
&\quad \mathsf{freevars}\ ce\ \cup\ \mathsf{boundvars}\ ce \subseteq ns\ \ \land\ \ \mathsf{wf}\ ce \\
&\qquad \Rightarrow\ \ l \Vdash (\mathsf{desugar}\ ce) \rightsquigarrow (\mathsf{desugar}\ ce')
\end{aligned}$$

That is, after desugaring compiler expressions into semantic expressions (desugar, see § 3), the action of the inliner for input ce, memory m, and name set ns lies within $\rightsquigarrow$ for some stacked **let**s l when the following hold:

– (memory_rel) m and l contain the same definitions, and each such definition *both* satisfies wf below *and* has bound/free variables within ns;
– (barendregt) bound names in ce are unique, and disjoint from free names;
– the bound variables of ce do not shadow (are disjoint from, #) any variables with known definitions, *i.e.*, those in the domain of m;
– all bound/free variables of ce are within ns; and
– (wf) ce is well-formed.

Proof outline. Induction over the implementation function inline. For each case of the proof, we apply rules of $\rightsquigarrow$ to justify each atomic inlining operation. □

Theorem 4. Top-level correctness of $\mathsf{inliner}$.

$$\vdash \mathsf{wf}\ ce\ \wedge\ \mathsf{closed}\ ce\ \Rightarrow\ (\mathsf{desugar}\ ce)\ \cong\ \big(\mathsf{desugar}\ (\mathsf{inliner}_h^k\ ce)\big)$$

Proof outline. Composition of Theorem 3 above with Theorem 2 (pg. 9), the soundness theorem for $\rightsquigarrow$. Unfolding the definition of $\mathsf{inliner}$, we use the soundness theorem of $\mathsf{freshen}$, the closed assumption, and the application of inline to empty memory $\varnothing$ to discharge the preconditions on Theorem 3. □

7 Integration into the PureCake Compiler

We insert the inliner and its associated cleanup of dead **let**-bindings as PureLang-to-PureLang transformations early in the PureCake compiler. In particular, directly after parsing and binding group analysis, as shown in Figure 2. Elimination of dead **let**s happens directly afterwards.

Unusually, the inliner runs before type inference. Ideally, it would take place afterwards: it changes program structure significantly, and type inference should execute on code resembling user input to allow direct error-reporting. The reasoning behind this design choice is PureCake's demand analysis, which facilitates strictness optimisations by annotating variables that can be evaluated eagerly. We found that running the inliner before demand analysis produces significantly better performance (§ 8, Figure 4). However, the soundness proof for demand analysis requires it to receive only well-typed input code. To run the inliner after type inference and before demand analysis, we would have to prove that it preserves well-typing, which is a significant undertaking due to PureLang's untyped AST. Future iterations of PureLang's AST are intended to be typed; therefore, we could consider proving type preservation in future work.

To update PureCake's compiler correctness theorem after integrating our inliner, we must establish that the inliner preserves both semantics and various syntactic invariants. We have already presented our proof of semantics preservation in § 6. The latter syntactic invariants guarantee that compiler expressions are closed and satisfy well-formedness properties which are checked as part of parsing. For example, PureLang forbids degenerate function applications to zero arguments: this can be expressed in the AST for PureLang compiler expressions but is ill-formed. Establishing preservation of the invariants is mostly mechanical, but quite tedious and long-winded.

8 Benchmarks

In this section we measure the efficacy of our inliner. In particular, we benchmark code generated by PureCake to determine how much the addition of the inliner improves runtime and memory overhead.

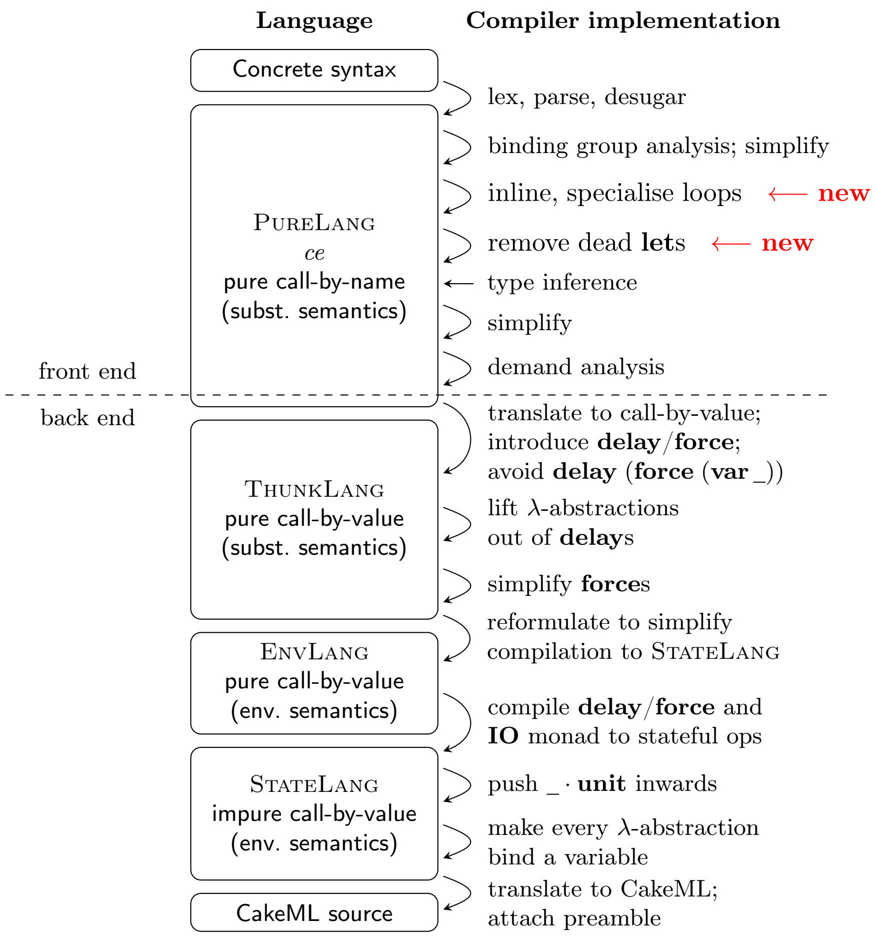


Fig. 2. High-level structure of the PureCake compiler. The inliner and its associated clean up are PureLang-to-PureLang passes which take place immediately after binding group analysis and before type inference.

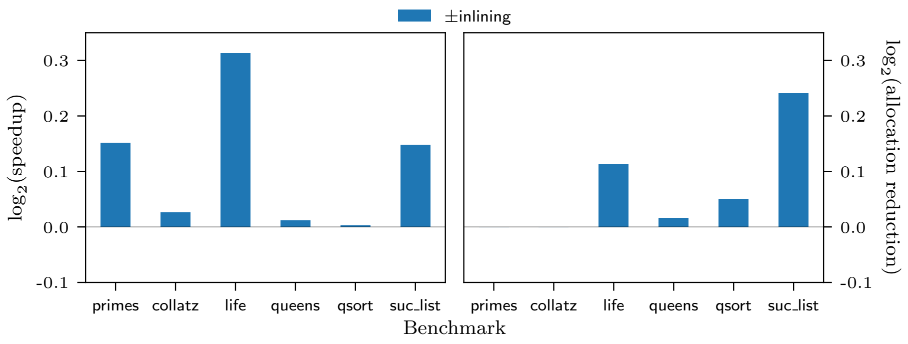


Fig. 3. Graphs showing the performance impact of our inliner: the base-2 logarithm of a ratio of measurements (execution time or heap allocations) with/without the inliner enabled: $\log_2\left({}^{m_{\text{disabled}}}/_{m_{\text{enabled}}}\right)$. Error bars are too small to be visible.

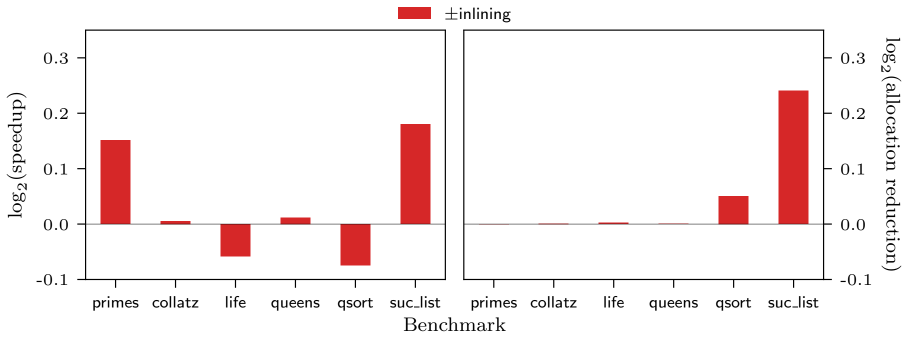


Fig. 4. Graphs showing the performance impact of our inliner when executed *after* PureCake's demand analysis. Performance is clearly worse compared to Figure 3; *therefore we do not pursue this approach.*

Methodology. We evaluate the performance of several benchmark programs with and without the inliner enabled, using an Intel® Xeon® E-2186G and 64 GB RAM. We consider the same programs as presented by the PureCake developers in prior work [10, §7.1]. We also add a new suc_list program, which repeatedly applies the `suc_list` function shown in § 1 to a list of natural numbers. Like the PureCake developers, we measure wall-clock runtime and total heap allocations as reported by the CakeML runtime. Our measurements are facilitated by existing benchmarking scripts found in the PureCake development.

Results. Figure 3 shows our results, plotted as two bar graphs: the left shows runtime speedup, the right shows allocation reduction. In many cases, our inliner significantly improves performance; in all cases it does not worsen performance. The value for each plot is obtained by taking the base-2 logarithm of a ratio: the measurement without the inliner enabled (*i.e.*, the longer duration or greater allocation) divided by the measurement with the inliner enabled. Expressed as

Table 1. Line counts for each part of our development.

Part of development	kLoC
Syntactic relation ($\leadsto$) and its soundness (§ 4)	2.6
Equational theory behind specialisation (§ 5)	4.0
Implementation of inliner (incl. specialisation) (§ 6.2)	0.6
Correctness of the implementation (§ 6.3)	3.7
Freshening and its correctness proof	3.1
Elimination of dead lets and its correctness proof	0.5
Total	**∼15**

a percentage, the most significant improvements are: a ∼20% reduction in the runtime of life, a ∼15% reduction in the allocations of suc_list.

Inliner placement. We noted in § 7 that our inliner should run before PureCake's demand analysis. Here, we justify that design choice. In particular, we benchmark a version of the PureCake compiler which runs our inliner directly *after* demand analysis. The results are shown in Figure 4. The improvements in runtime and memory overhead are reduced for several benchmarks, and in some cases runtime even worsens overall. Therefore, our inliner should run before demand analysis for maximum benefit.

Code size and compile times. Simple measurements of code size show that our inliner can produce significantly larger CakeML programs (∼50% increase); however CakeML's efficient handling of inserted **let**s reduces the effect for binaries (< 15% overall increase). Compile times are unaffected: these remain dominated by PureCake's type-checking and CakeML's register allocation.

Line counts. Our work adds to PureCake significantly. Table 1 shows line counts for each part of our development, measured using `wc -l`.

9 Related Work

Verified inlining in functional languages. CakeML [12] compiles a subset of Standard ML (strict, impure) to several mainstream architectures with end-to-end guarantees. It performs function inlining in its second intermediate language, ClosLang, which has first-class closures. A flow analysis discovers invocations of known functions, and simultaneously inlines closed functions which themselves do not contain closures. Use of de Bruijn indices sidesteps reasoning about shadowing and freshening. As in our work, recursive applications of inlining improve the performance of higher-order functions; we go one step further with specialisation and the inlining of open terms which can contain λ-abstractions.

CertiCoq [2] verifiably compiles Gallina (the metalanguage of Coq) to C light, an intermediate language early in CompCert's pipeline. One of its passes [4] performs several *shrink reductions* simultaneously: transformations that only reduce code size. One such reduction is the inlining of functions which are applied exactly once; in this case, inlining *is* β-reduction, contrary to our discussion in § 4.1. Restriction to shrink reductions further removes the need for a recursion limit as code size strictly decreases on each recursive call. Their verification relies on a more general rewrite system which permits inlining of functions which are used multiple times. A separate pass [16] further inlines small non-recursive functions which can be applied multiple times; here a key concern is maintenance of A-normal form expressions. In all proofs, the Barendregt variable convention (*i.e.*, barendregt) is used to avoid name clashes.

Pilsner [15] compiles a strict impure language to an idealised assembly, inlining select top-level functions in its intermediate representation. Recursive functions can be unrolled in this way, but not specialised. Again, the Barendregt variable convention is enforced. The focus here is on the novel proof technique of parametric inter-language simulations (PILS) to enable compositional compiler correctness, where PureCake focuses on mechanised whole-program compiler correctness for a realistic language.

Other verified inlining passes. CompCert [13] compiles a subset of C99, performing function inlining in its register transfer language (RTL). This control flow graph (CFG) representation differs considerably from the functional PureLang; inlining considers only top-level function declarations in the RTL setting. Rather than using a recursion limit, CompCert guarantees termination by forbidding inlining of functions within their own bodies.

CompCert also performs *lazy code motion* [19] within RTL. A special case of this transformation is loop-invariant code motion, which loosely resembles our specialisation: both are concerned with moving constant expressions out of loops, but in our functional setting loops are expressed as recursive functions. Their verification uses translation validation [18]: an unverified tool transforms code, and then per-run automation proves that semantics has been preserved.

The Plutus Tx language from the Cardano blockchain platform resembles a subset of Haskell, and is compiled to a custom language known as Plutus Core. The compiler is implemented as a GHC plugin: GHC machinery first lowers Plutus Tx to a System F-like language, which is then optimised and compiled further. The compiler is verified using *translation certification* [11], which aims to make translation validation approaches less brittle by combining automated and manual proof. As in PureCake, syntactic relations are used to encapsulate semantics-preserving transformations: automated proof shows that unverified code transformations inhabit the relations, and manual proof shows that the relations preserve semantics. Translation certification is robust to evolving compiler implementations because the syntactic proofs are more amenable to automated verification than the semantic ones. A syntactic relation akin to § 4 justifies inlining; however, semantic verification is ongoing work at the time of writing. The Barendregt variable convention is enforced in this work too.

Verified optimisation of realistic Haskell-like languages. The CoreSpec project[4] tackles verified variants of Haskell as implemented by GHC. For example, GHC's dependent types extensions were proposed using formal specifications of the syntax, semantics, and typing rules of GHC's Core language [20]. The unverified tool `hs-to-coq` [6] translates Haskell code to Gallina (Coq's metalanguage), leveraging Coq's logic to enable equational reasoning about real-world programs. A future aim of the project is to derive Coq models of Core automatically from GHC's implementation, prove correctness of optimisations within Coq, and integrate the resulting verified code back into GHC as a plugin. Where CoreSpec focuses on accurate modelling of GHC with the loss of some trust, PureCake instead sacrifices faithfulness for end-to-end guarantees.

GHC's arity analysis pass [5] η-expands functions to avoid excessive thunk allocations. Its mechanised proof of correctness for a simplified Core language relies on an explicitly call-by-need semantics to show performance preservation, *i.e.*, that η-expansion does not reduce value-sharing.

10 Summary and Future Work

This paper has described our work on a verified inlining and loop specialisation pass for PureLang, a lazy functional programming language. First, we verified a syntactic relation which defines an envelope of permitted inlining transformations, independent of heuristic choices. We used a novel phrasing of inlining as the pushing in and pulling out of `let`-bindings to prove the relation sound using PureLang's equational theory. Our inliner implementation is then proven to remain within this envelope. We have integrated our work into the PureCake compiler, an end-to-end verified compiler, and demonstrated significant performance improvements. To the best of our knowledge, ours is the first verified function inliner for a lazy functional programming language, and the first verified loop specialiser for any functional language.

In future work, we intend to support loop unrolling and develop better heuristics that decide when to do inlining. Loop unrolling will probably involve augmenting the definition of **lets** so that it can hold both **let** expressions and **letrec**s. Developing good heuristics will require some careful experimentation with the compiler implementation. We do not expect adjustment to the inliner's heuristics to impact our correctness proofs in any significant way, since the proofs are designed to be independent of heuristic choices.

Acknowledgements. Hrutvik Kanabar was supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS). Magnus Myreen was supported by Swedish Research Council grant 2021-05165.

Data availability statement. An artifact supporting the results presented in this paper is openly available on Zenodo [9]. The latest development version of PureCake is available on GitHub (https://github.com/cakeml/pure).

[4] https://deepspec.org/entry/Project/Haskell+CoreSpec

References

1. Abramsky, S.: The lazy λ-calculus. In: Research Topics in Functional Programming. Addison Wesley (1990)
2. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.Z.: CertiCoq: A verified compiler for Coq. In: Workshop on Coq for Programming Languages (CoqPL) (2017), https://popl17.sigplan.org/details/main/9/CertiCoq-A-verified-compiler-for-Coq
3. Barendregt, H.P.: The lambda calculus - its syntax and semantics, Studies in logic and the foundations of mathematics, vol. 103. North-Holland (1985)
4. Bélanger, O.S., Appel, A.W.: Shrink fast correctly! In: Principles and Practice of Declarative Programming (PPDP). ACM (2017). https://doi.org/10.1145/3131851.3131859
5. Breitner, J.: Formally proving a compiler transformation safe. In: Symposium on Haskell. ACM (2015). https://doi.org/10.1145/2804302.2804312
6. Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). Proc. ACM Program. Lang. **2**(ICFP) (2018). https://doi.org/10.1145/3236784
7. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Inf. Comput. **124**(2) (1996). https://doi.org/10.1006/inco.1996.0008
8. Hughes, J.: Why functional programming matters. Comput. J. **32**(2) (1989). https://doi.org/10.1093/comjnl/32.2.98
9. Kanabar, H., Korban, K., Myreen, M.O.: Artifact for "Verified Inlining and Specialisation for PureCake" (2024). https://doi.org/10.5281/zenodo.10456887
10. Kanabar, H., Vivien, S., Abrahamsson, O., Myreen, M.O., Norrish, M., Åman Pohjola, J., Zanetti, R.: PureCake: A verified compiler for a lazy functional language. In: Programming Language Design and Implementation (PLDI). ACM (2023). https://doi.org/10.1145/3591259
11. Krijnen, J.O.G., Chakravarty, M.M.T., Keller, G., Swierstra, W.: Translation certification for smart contracts. In: Functional and Logic Programming (FLOPS). Lecture Notes in Computer Science, vol. 13215. Springer (2022). https://doi.org/10.1007/978-3-030-99461-7_6
12. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Programming Languages (POPL). ACM (2014). https://doi.org/10.1145/2535838.2535841
13. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009). https://doi.org/10.1145/1538788.1538814
14. Myreen, M.O.: The CakeML project's quest for ever stronger correctness theorems (invited paper). In: Interactive Theorem Proving (ITP). LIPIcs, vol. 193. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.1
15. Neis, G., Hur, C., Kaiser, J., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: a compositionally verified compiler for a higher-order imperative language. In: International Conference on Functional Programming (ICFP). ACM (2015). https://doi.org/10.1145/2784731.2784764
16. Paraskevopoulou, Z., Li, J.M., Appel, A.W.: Compositional optimizations for CertiCoq. Proc. ACM Program. Lang. **5**(ICFP) (2021). https://doi.org/10.1145/3473591

17. Peyton Jones, S.L., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming **12** (2002)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1384. Springer (1998). https://doi.org/10.1007/BFb0054170
19. Tristan, J., Leroy, X.: Verified validation of lazy code motion. In: Programming Language Design and Implementation (PLDI). ACM (2009). https://doi.org/10.1145/1542476.1542512
20. Weirich, S., Voizard, A., de Amorim, P.H.A., Eisenberg, R.A.: A specification for dependent types in Haskell. Proc. ACM Program. Lang. **1**(ICFP) (2017). https://doi.org/10.1145/3110275

Suspension Analysis and Selective Continuation-Passing Style for Universal Probabilistic Programming Languages

Daniel Lundén[1] (✉), Lars Hummelgren[2], Jan Kudlicka[3], Oscar Eriksson[2], and David Broman[2,4]

[1] Oracle, Stockholm, Sweden, `daniel.lunden@oracle.com`
[2] EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden, `{larshum,oerikss,dbro}@kth.se`
[3] Department of Data Science and Analytics, BI Norwegian Business School, Oslo, Norway, `jan.kudlicka@bi.no`
[4] Computer Science Department, Stanford University, California, USA `broman@stanford.edu`

Abstract. Universal probabilistic programming languages (PPLs) make it relatively easy to encode and automatically solve statistical inference problems. To solve inference problems, PPL implementations often apply Monte Carlo inference algorithms that rely on execution suspension. State-of-the-art solutions enable execution suspension either through (i) continuation-passing style (CPS) transformations or (ii) efficient, but comparatively complex, low-level solutions that are often not available in high-level languages. CPS transformations introduce overhead due to unnecessary closure allocations—a problem the PPL community has generally overlooked. To reduce overhead, we develop a new efficient selective CPS approach for PPLs. Specifically, we design a novel static suspension analysis technique that determines parts of programs that require suspension, given a particular inference algorithm. The analysis allows selectively CPS transforming the program only where necessary. We formally prove the correctness of the analysis and implement the analysis and transformation in the Miking CorePPL compiler. We evaluate the implementation for a large number of Monte Carlo inference algorithms on real-world models from phylogenetics, epidemiology, and topic modeling. The evaluation results demonstrate significant improvements across all models and inference algorithms.

Keywords: Probabilistic programming · Static analysis · Continuation-passing style.

1 Introduction

Probabilistic programming languages (PPLs), such as Anglican [50], Birch [36], WebPPL [18], Stan [10], Pyro [6], and Gen [11], make it possible to encode and solve statistical inference problems. Such inference problems are of significant interest in many research fields, including phylogenetics [43], computer vision [25], topic modeling [7], inverse graphics [20], and cognitive science [19]. A particularly appealing feature of PPLs is the separation between the inference problem specification (the language) and the inference algorithm used to solve the problem (the language implementation). This separation allows PPL users to focus solely on encoding their inference problems while inference algorithm experts deal with the intricacies of inference implementation.

Implementations of PPLs apply many different inference algorithms. Monte Carlo inference algorithms—such as Markov chain Monte Carlo (MCMC) [16] and sequential Monte Carlo (SMC) [12]—are popular due to their asymptotic correctness and relative ease of implementation for *universal*[5] PPLs. The central idea behind all Monte Carlo methods in PPLs is to execute probabilistic programs multiple times to generate *samples* that approximate the target distribution for the encoded inference problem. However, repeated execution is expensive, and PPL implementations must avoid unnecessary overhead.

Monte Carlo algorithms often need to *suspend* executions. For example, MCMC algorithms can suspend at *random draws* in the program to avoid unnecessary re-execution when proposing new executions, and SMC algorithms can suspend at *likelihood updates* to *resample* executions. Languages such as WebPPL [18] and Anglican [50], and the approach described by Ritchie et al. [41], apply *continuation-passing style (CPS)* transformations [3] to enable arbitrary suspension during execution. The main benefit of CPS transformations is that they are relatively easy to implement in functional programming languages. However, one disadvantage with CPS transformations is that high-performance low-level languages, without higher-order functions, do not support them. For this reason, there are also more direct low-level alternatives to CPS, including non-preemptive multitasking (e.g., coroutines [15]) and PPL control-flow graphs [30]. These more direct alternatives can additionally avoid much of the overhead resulting from CPS[6], but are more complex to implement.

We consider how to bridge the performance gap between CPS-based PPLs and lower-level PPLs that rely on, e.g., direct implementation of coroutines. We consider optimizations at the CPS transformation level, and not the translation from CPS-based PPLs to lower-level representations. CPS overhead is a result of closure allocations for continuations. We make the important observation that PPLs do not require the arbitrary suspensions provided by full CPS transformations. Most Monte Carlo inference algorithms require suspension only in very specific parts of programs. Current state-of-the-art CPS-based PPLs do not consider inference-specific suspension requirements to reduce CPS overhead.

We design a new static suspension analysis and a new selective CPS transformation for PPLs that together significantly reduce runtime overhead com-

[5] A term that first appeared in Goodman et al. [17], indicating expressive PPLs where the number and types of random variables are not always known statically.

[6] Note that CPS only results in overhead if programs reify the continuations at runtime to, e.g., suspend computations. Traditional CPS-based compilers often only use CPS as an intermediate form during compilation, which does not result in runtime overhead.

pared to a traditional full CPS transformation. Current state-of-the-art functional PPLs that use CPS for execution suspension can therefore greatly benefit from our new approach. The suspension analysis identifies all parts of programs that may require suspension as a result of applying a particular inference algorithm. We formalize the suspension analysis algorithm using a core PPL calculus equipped with a big-step operational semantics. Specifically, the challenge lies in capturing how suspension requirements propagate through the program in the presence of higher-order functions. Furthermore, we formalize the selective CPS transformation and justify its correctness when guided by the suspension analysis. Prior work on selective CPS for general-purpose programming languages, e.g., by Nielsen [38] and Asai and Uehara [4], focuses on analyses based on type systems and type inference. In contrast, we instead build our suspension analysis using 0-CFA [46] and it operates directly on an untyped calculus.

Overall, we (i) prove that the suspension analysis is correct, (ii) show that the resulting selective CPS transformation gives significant performance gains compared to using a full CPS transformation, and (iii) show that the overall approach is directly applicable to a large set of inference algorithms. Specifically, we evaluate the approach for the following inference algorithms: likelihood weighting, the SMC bootstrap particle filter, the SMC alive particle filter [24], aligned lightweight MCMC [29,49], and particle-independent Metropolis–Hastings [40]. We consider each inference algorithm for four real-world models from phylogenetics, epidemiology, and topic modeling.

We implement the suspension analysis and selective CPS transformation in Miking CorePPL [30,9]. Similarly to WebPPL and Anglican, the implementation supports the co-existence of many inference problems and applications of inference algorithms to these problems within the same program. However, compared to full CPS, such programs are more challenging to handle with selective CPS, as the CPS transformation of an inference problem also depends on the applied inference algorithm—different inference algorithms generally require different suspensions. To complicate things further, different inference problems may share some code, or the PPL user may apply two different inference algorithms to the same inference problem. The compiler must then apply different CPS transformations to different parts of the program, and sometimes even many different CPS transformations to separate copies of the *same* part of the program. To solve this, we develop an approach that, for any given Miking CorePPL program, *extracts* all possible inference problems and corresponding inference algorithm applications. This extraction procedure allows the correct application of selective CPS throughout the program.

In summary, we make the following contributions.

- We design, formalize, and prove the correctness of a suspension analysis for PPLs, where the suspension requirements come from a given inference algorithm (Section 4).
- We design and formalize a new selective CPS transformation for PPLs. Compared to full CPS, selectively CPS transforming PPL programs guided by

the suspension analysis significantly reduces runtime overhead resulting from unnecessary closure allocations (Section 5).
- We implement the suspension analysis and selective CPS transformation in the Miking CorePPL compiler. Unlike full CPS, selective CPS introduces challenges for probabilistic programs containing many inference problems and inference algorithm applications. We implement an approach that correctly applies selective CPS to such programs by extracting individual inference problems (Section 6).

Section 7 presents the evaluation and its results for the implementations in Miking CorePPL, Section 8 discusses related work in more detail, and Section 9 concludes. We first consider a motivating example in Section 2 and introduce the underlying PPL calculus in Section 3.

An extended version of the paper is available at arXiv [31]. We use the † symbol in the text to indicate that more information (e.g., proofs) is available in the extended version.

2 A Motivating Example

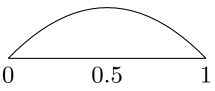
This section introduces the running example in Fig. 1 and uses it to present the basic idea behind PPLs and how inference algorithms such as SMC and MCMC make use of CPS to suspend executions. Most importantly, we illustrate the motivation and key ideas behind selective CPS for PPLs.

Consider the probabilistic program in Fig. 1a, written in a functional-style PPL. The program encodes an inference problem for estimating the probability distribution over the bias of a coin, *conditioned* on the outcome of four experimental coin flips: true, true, false, and true (true = heads and false = tails). At line 1, we use the PPL-specific `assume` construct to define our *prior* belief in the bias a_1 of the coin. We set this prior belief to a Beta$(2, 2)$ probability distribution, illustrated in Fig. 1b. In the illustration, 0 indicates a coin that always results in false, 1 a coin that always results in true, and 0.5 a fair coin. We see that our prior belief is quite evenly spread out, but with more probability mass towards a fair coin. To condition this prior distribution on the observed coin flips, we conceptually execute the program in Fig 1a infinitely many times, *sampling* values from the prior Beta distribution at `assume` (line 1) and, as a side effect, *accumulating the product of weights* given as argument to the PPL-specific `weight` construct (line 4). We make the four consecutive calls `weight` `(`$f_{\text{Bernoulli}}$ a_1 true`)`, `weight` `(`$f_{\text{Bernoulli}}$ a_1 true`)`, `weight` `(`$f_{\text{Bernoulli}}$ a_1 false`)`, and `weight` `(`$f_{\text{Bernoulli}}$ a_1 true`)`[7], using the recursive function *iter*. The function application $f_{\text{Bernoulli}}$ a_1 o gives the probability of the outcome o given a bias a_1 for the coin. I.e., $f_{\text{Bernoulli}}$ a_1 true $= a_1$ and $f_{\text{Bernoulli}}$ a_1 false $= 1 - a_1$. So, for example, a sample $a_1 = 0.4$ gets the accumulated weight $0.4 \cdot 0.4 \cdot 0.6 \cdot 0.4$

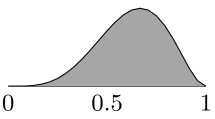
[7] PPLs also commonly use a similar built-in function `observe` to update the weight. For example, `observe` (Bernoulli a_1) true is equivalent to `weight` `(`$f_{\text{Bernoulli}}$ a_1 true`)`.

```
1 let a1 = assume (Beta 2 2) in
2 let rec iter = λobs.
3   if null obs then () else
4     weight (f_Bernoulli a1 (head obs));
5     iter (tail obs)
6 in
7 iter [true,true,false,true];
8 a1
```

(a) Program $\mathbf{t}_{\text{example}}$.



(b) Beta(2,2).



(c) Distribution of $\mathbf{t}_{\text{example}}$.

```
Suspension_assume(Beta 2 2, λa1.
  let rec iter = λobs.
    if null obs then () else
      weight (f_Bernoulli(a1)
                (head obs));
      iter (tail obs)
  in
  iter [true,true,false,true];
  a1)
```

(d) Suspension at `assume`.

```
let a1 = assume (Beta 2 2) in
let rec iter = λk. λobs.
  if null obs then k ()
  else
    Suspension_weight(
      f_Bernoulli(a1) (head obs),
      (λ_. iter k (tail obs)))
in
iter (λ_. a1)
  [true,true,false,true];
```

(e) Suspension at `weight`.

```
let k7 = λt6.
 let k8 = λt7.
  Suspension_assume(t7, λa1.
   let rec iter = λk1. λobs.
    let k2 = λt1.
     if t1 then k1 () else
      let k3 = λt2.
       let k4 = λt3.
        let k5 = λt4.
         Suspension_weight(t4, λ_.
          let k6 = λt5. iter k1 t5 in
          tail_CPS k6 obs)
        in t2 k5 t3
       in head_CPS k4 obs
      in f_Bernoulli_CPS k3 a1
    in null_CPS k2 obs
   in iter (λ_. a1)
         [true,true,false,true])
 in t6 k8 2
in Beta_CPS k7 2
```

(f) Full CPS.

Fig. 1: A probabilistic program $\mathbf{t}_{\text{example}}$ modeling the bias of a coin. Fig. (a) gives the program. The function $f_{\text{Bernoulli}}$ is the probability mass function of the Bernoulli distribution. Fig. (b) illustrates the distribution for a_1 at line 1 in (a). Fig. (c) shows the set of (weighted) samples resulting from conceptually running $\mathbf{t}_{\text{example}}$ infinitely many times. Fig. (d) and Fig. (e) show the selective CPS transformations required for suspension at `assume` and `weight`, respectively. Fig. (f) gives $\mathbf{t}_{\text{example}}$ in full CPS, with suspensions at `assume` and `weight`. The $_{\text{CPS}}$ subscript indicates CPS-versions of intrinsic functions such as *head* and *tail*.

and $a_1 = 0.7$ the accumulated weight $0.7 \cdot 0.7 \cdot 0.3 \cdot 0.7$. The end result is an infinite set of *weighted* samples of a_1 (the program returns a_1 at line 8) that approximate the *posterior* or *target* distribution of Fig. 1a, illustrated in Fig 1c. Note that, because we observed three true outcomes and only one false, the weights shift the probability mass towards 1 and narrows it slightly as we are now more sure about the bias of the coin. Increasing the number of experimental coin flips would make Fig. 1c more and more narrow.

We can approximate the infinite number of samples by running the program a large (but finite) number of times. This basic inference algorithm is known as *likelihood weighting*. The problem with likelihood weighting is that it is only accurate enough for simple models. For complex models, it is common that only a few likelihood weighting samples (often only one) get much larger weights relative to the other samples, greatly reducing inference accuracy. Real-world models require more powerful inference algorithms based on, e.g., SMC or MCMC. A key requirement in both SMC and MCMC is the ability to *suspend* executions of probabilistic programs at calls to `weight` and/or `assume`. One way to enable suspensions is by writing programs in CPS. We first illustrate a simple use of CPS to suspend at `assume` in Fig. 1d. Here, the program immediately returns an object $\text{Suspension}_{\texttt{assume}}(\text{Beta 2 2}, k)$, indicating that execution stopped at an `assume` with the argument Beta 2 2 and a continuation k (i.e., the abstraction binding a_1) that executes the remainder of the program. With likelihood weighting, we would simply sample a value a_1 from the Beta 2 2 distribution and resume execution by calling $k\ a_1$. This call then runs the program until termination and results in the actual return value of the program, which is a_1. Many MCMC inference algorithms reuse samples from previous executions at $\text{Suspension}_{\texttt{assume}}$, and the suspensions are thus useful to avoid unnecessary re-execution [41].

As a second example, we illustrate suspension at `weight` for, e.g., SMC inference in Fig. 1e. Here, we require suspensions in the middle of the recursive call to *iter*, and writing the program in CPS is more challenging. We rewrite the *iter* function to take a continuation k as argument, and call the continuation with the return value () at line 3 instead of directly returning () as in Fig. 1a at line 3. This continuation argument k is precisely what allows us to construct and return $\text{Suspension}_{\texttt{weight}}$ objects at line 5. To illustrate the suspensions, consider executing the program with likelihood weighting. First, the program returns the object $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k')$, where k' is the continuation that line 7 constructs. Likelihood weighting now updates the weight for the execution with the value $f_{\text{Bernoulli}(a_1)}$ true and resumes execution by calling k' (). Similarly, this next execution returns $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k'')$ for the second recursive call to *iter*, and we again update the weight and resume by calling k'' (). We similarly encounter $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{false}, k''')$ and $\text{Suspension}_{\texttt{weight}}(f_{\text{Bernoulli}(a_1)}\ \text{true}, k'''')$ before the final call k'''' () runs the program to termination and produces the actual return value a_1. In SMC, we run many executions concurrently and wait until they all have returned a $\text{Suspension}_{\texttt{weight}}$ object. At this point, we resample the executions according to their weights (the first value in $\text{Suspension}_{\texttt{weight}}$), which discards executions with low weight and replicates executions with high weight. After resampling, we continue to the next suspension and resampling by calling the continuations.

PPL implementations enable suspensions at `assume` and/or `weight` through automatic and full CPS transformations. Fig. 1f illustrates such a transformation for Fig. 1a. We indicate CPS versions of intrinsic functions with the $_{\text{CPS}}$ subscript. Note that the full CPS transformation results in many additional closure allocations compared to Fig. 1d and Fig. 1e. As a result, runtime overhead increases significantly. The contribution in this paper is a static analysis that allows an automatic and selective CPS transformation of programs, as in Fig. 1d and Fig. 1e. With a selective transformation, we avoid many unnecessary closure allocations, and can significantly reduce runtime overhead while still allowing suspensions as required for a given inference algorithm.

3 Syntax and Semantics

This section introduces the PPL calculus used to formalize the suspension analysis in Section 4 and selective CPS transformation in Section 5. Section 3.1 gives the abstract syntax and Section 3.2 a big-step operational semantics. Section 3.3 introduces A-normal form—a prerequisite for both the suspension analysis and the selective CPS transformation.

3.1 Syntax

We build upon the standard untyped lambda calculus, representative of functional universal PPLs such as Anglican, WebPPL, and Miking CorePPL. We define the abstract syntax below.

Definition 1 (Terms, values, and environments). *We define terms* $\mathbf{t} \in T$ *and values* $\mathbf{v} \in V$ *as*

$$\begin{aligned}
\mathbf{t} ::= {} & x \mid c \mid \lambda x.\ \mathbf{t} \mid \mathbf{t}\ \mathbf{t} \mid \texttt{let}\ x = \mathbf{t}\ \texttt{in}\ \mathbf{t} \qquad\qquad \mathbf{v} ::= c \mid \langle \lambda x.\ \mathbf{t}, \rho \rangle \\
& \mid \texttt{if}\ \mathbf{t}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{assume}\ \mathbf{t} \mid \texttt{weight}\ \mathbf{t} \\
& x, y \in X \quad \rho \in P \quad c \in C \quad \{\text{false}, \text{true}, ()\} \cup \mathbb{R} \cup D \subseteq C.
\end{aligned} \tag{1}$$

The countable set X contains variable names, C intrinsic values and operations, and $D \subset C$ intrinsic probability distributions. The set P contains evaluation environments*, i.e., maps from variables in X to values in V.*

Definition 2 (Target language terms). *As a target language for the selective CPS transformation in Section 5, we additionally extend Definition 1 to target language terms* $\mathbf{t} \in T^+$ *by*

$$\mathbf{t} \mathrel{+}= \text{Suspension}_{\texttt{assume}}(\mathbf{t}, \mathbf{t}) \mid \text{Suspension}_{\texttt{weight}}(\mathbf{t}, \mathbf{t}). \tag{2}$$

Fig. 1a gives an example of a term in T, and Fig. 1d and Fig. 1e of terms in T^+. However, note that the programs in Fig. 1 also use the list constructor `[...]` (not part of the above definitions) to make the example more interesting.

In addition to the standard variable, abstraction, and application terms in the untyped lambda calculus, we include explicit `let` expressions for convenience. Furthermore, we use the syntactic sugar `let rec` f `=` $\lambda x.\mathbf{t}_1$ `in` $\mathbf{t}_2$ to define recursive functions (translating to an application of a call-by-value fixed-point combinator). We use $\mathbf{t}_1$`;` $\mathbf{t}_2$ as a shorthand for `(`λ`_.`$\mathbf{t}_2$`)` $\mathbf{t}_1$, where _ means that we do not use the argument. That is, we evaluate $\mathbf{t}_1$ for side effects only.

We include a set C of intrinsic operations and constants essential to inference problems encoded in PPLs. The set of intrinsics includes boolean truth values, the unit value, real numbers, and probability distributions. We can also add further operations and constants to C. For example, we can let $+ \in C$ to support addition of real numbers. To allow control flow to depend on intrinsic values, we include `if` expressions that use intrinsic booleans as condition.

We saw examples of the `assume` and `weight` constructs in Section 2. The `assume` construct takes distributions $D \subset C$ as argument, and produces random variables distributed according to these distributions. For example, we can let $\mathcal{N} \in C$ be a function that constructs normal distributions. Then, `assume (`$\mathcal{N}$ `0 1)`, where $\mathcal{N}$ 0 1 $\in D$, defines a random variable with a standard normal distribution. Partially constructed distributions, e.g., $\mathcal{N}$ 0, are also in C, but not in D (they are not yet proper distributions). As we saw in Section 2, the `weight` construct updates the likelihood with the real number given as argument, and allows conditioning on data (e.g., the four coin flips in Fig. 1).

3.2 Semantics

We construct a call-by-value big-step operational semantics, based on Lundén et al. [29], describing how to evaluate terms $\mathbf{t} \in T$. Such a semantics is a key component when formally defining the probability distributions corresponding to terms $\mathbf{t} \in T$ (e.g., the distribution in Fig. 1c corresponding to the program in Fig. 1a) and also when proving various properties of PPLs and their inference algorithms (e.g., inference correctness). See, e.g., the work by Borgström et al. [8] and Lundén et al. [28] for full formal treatments.

We use the semantics to formally define suspension, and use this definition to state the soundness of the suspension analysis in Section 4 (Theorem 1). We use a big-step semantics, as we do not require the additional control provided by a small-step semantics. For example, we do not concern ourselves with details of termination, as the soundness of the analysis relates only to terminating executions. Fig. 2 presents the full semantics as a relation $\rho \vdash \mathbf{t} \;{}^{s}{\Downarrow}_{u}^{w}\; \mathbf{v}$ over tuples $(P, T, S, \{\text{false}, \text{true}\}, \mathbb{R}, V)$. S is a set of *traces* capturing the random draws at `assume` during evaluation. Intuitively, $\rho \vdash \mathbf{t} \;{}^{s}{\Downarrow}_{u}^{w}\; \mathbf{v}$ holds iff $\mathbf{t}$ evaluates to $\mathbf{v}$ in the environment ρ with the trace s and the total probability density (i.e., the accumulated weight) w. We describe the suspension flag u later in this section.

Most of the rules are standard and we focus on explaining key properties related to PPLs and suspension. We first consider the rule (Const-App), which uses the δ-function to evaluate intrinsic operations.

$$\frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{u_1} \langle \lambda x.\mathbf{t}, \rho' \rangle \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{u_2} \mathbf{v}_2 \quad \rho', x \mapsto \mathbf{v}_2 \vdash \mathbf{t} \;{}^{s_3}\!\Downarrow^{w_3}_{u_3} \mathbf{v}}{\rho \vdash \mathbf{t}_1\ \mathbf{t}_2 \;{}^{s_1\|s_2\|s_3}\!\Downarrow^{w_1\cdot w_2\cdot w_3}_{u_1\vee u_2\vee u_3} \mathbf{v}}\,(\textsc{App})$$

$$\frac{}{\rho \vdash x \;{}^{[]}\!\Downarrow^{1}_{\text{false}} \rho(x)}\,(\textsc{Var}) \qquad \frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{u_1} c_1 \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{u_2} c_2}{\rho \vdash \mathbf{t}_1\ \mathbf{t}_2 \;{}^{s_1\|s_2}\!\Downarrow^{w_1\cdot w_2}_{u_1\vee u_2} \delta(c_1, c_2)}\,(\textsc{Const-App})$$

$$\frac{}{\rho \vdash \lambda x.\mathbf{t} \;{}^{[]}\!\Downarrow^{1}_{\text{false}} \langle \lambda x.\mathbf{t}, \rho \rangle}\,(\textsc{Lam}) \qquad \frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{u_1} \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{u_2} \mathbf{v}}{\rho \vdash \texttt{let}\ x = \mathbf{t}_1\ \texttt{in}\ \mathbf{t}_2 \;{}^{s_1\|s_2}\!\Downarrow^{w_1\cdot w_2}_{u_1\vee u_2} \mathbf{v}}\,(\textsc{Let})$$

$$\frac{}{\rho \vdash c \;{}^{[]}\!\Downarrow^{1}_{\text{false}} c}\,(\textsc{Const}) \qquad \frac{\rho \vdash \mathbf{t}_1 \;{}^{s_1}\!\Downarrow^{w_1}_{u_1} \text{true} \quad \rho \vdash \mathbf{t}_2 \;{}^{s_2}\!\Downarrow^{w_2}_{u_2} \mathbf{v}_2}{\rho \vdash \texttt{if}\ \mathbf{t}_1\ \texttt{then}\ \mathbf{t}_2\ \texttt{else}\ \mathbf{t}_3 \;{}^{s_1\|s_2}\!\Downarrow^{w_1\cdot w_2}_{u_1\vee u_2} \mathbf{v}_2}\,(\textsc{If-True})$$

$$\frac{\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u} d \quad w' = f_d(c)}{\rho \vdash \texttt{assume}\ \mathbf{t} \;{}^{s\|[c]}\!\Downarrow^{w\cdot w'}_{suspend_{\texttt{assume}}\vee u} c}\,(\textsc{Assume}) \qquad \frac{\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u} w'}{\rho \vdash \texttt{weight}\ \mathbf{t} \;{}^{s}\!\Downarrow^{w\cdot w'}_{suspend_{\texttt{weight}}\vee u} ()}\,(\textsc{Weight})$$

Fig. 2: A big-step operational semantics for $\mathbf{t} \in T$. We omit the rule (If-False) for brevity; it is analogous to (If-True). The environment $\rho, x \mapsto \mathbf{v}$ denotes ρ extended with a binding $\mathbf{v}$ for x. For each $d \in D$, the function f_d is its probability density or probability mass function. E.g., $f_{\mathcal{N}(0,1)}(x) = e^{x^2/2}/\sqrt{2\pi}$, the density function of the standard normal distribution. We use the following notation: $\|$ for sequence concatenation, $\cdot$ for multiplication, and $\vee$ for logical disjunction.

Definition 3 (Intrinsic arities and the δ-function). *For each $c \in C$, we let $|c| \in \mathbb{N}$ denote its* arity*. We also assume the existence of a partial function $\delta : C \times C \to C$ such that if $\delta(c, c_1) = c_2$, then $|c| > 0$ and $|c_2| = |c| - 1$.*

For example, $\delta((\delta(+, 1)), 2) = 3$. We use the arity property of intrinsics to formally define traces.

Definition 4 (Traces). *For all $s \in S$, s is a sequence of intrinsics with arity 0, called a* trace*. We write $s = [c_1, c_2, \dots, c_n]$ to denote a trace s with n elements.*

The rule (Assume) formalizes random draws and consumes elements of the trace. Specifically, (Assume) updates the evaluation's total probability density $w \in \mathbb{R}$ with the density w' of the first trace element with respect to the distribution given as argument to `assume`. The rule (Weight) furthermore directly modifies the total probability density according to the `weight` argument.

We now consider the special suspension flag u in the derivation $\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u} \mathbf{v}$.

Definition 5 (Suspension requirement). *A derivation $\rho \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u} \mathbf{v}$* requires suspension *if the suspension flag u is true.*

For example, the rule (App) requires suspension if $u_1 \vee u_2 \vee u_3$—i.e., if any subderivation requires suspension. To reflect the particular suspension requirements in SMC and MCMC inference, we limit the source of suspension requirements to `assume` and `weight`. We turn the individual sources on and off through the

```
let t1 = 2 in
let t2 = 2 in
let t3 = Beta in
let t4 = t3 t1 in
let t5 = t4 t2 in
let a1 = assume t5 in
let rec iter = λobs.
  let t6 = null in
  let t7 = t6 obs in
  let t8 =
    if t7 then
      let t9 = () in
      t9
    else
      let t10 = f_Bernoulli in
      let t11 = t10 a1 in
      let t12 = head in
      let t13 = t12 obs in
      let t14 = t11 t13 in
      let w1 = weight t14 in
      let t15 = tail in
      let t16 = t15 obs in
      let t17 = iter t16 in
      t17
  in
  t8
in
let t18 = true in
let t19 = false in
let t20 = true in
let t21 = true in
let t22 = [t21,t20,t19,t18] in
let t23 = iter t22 in
a1
```

Fig. 3: The running example $\mathbf{t}_{\text{example}}$ from Fig. 1a transformed to ANF.

boolean variables $suspend_{\texttt{assume}}$ and $suspend_{\texttt{weight}}$ in Fig. 2. For the examples in the remainder of this paper, we let $suspend_{\texttt{weight}} = \text{true}$ and $suspend_{\texttt{assume}} = \text{false}$ (i.e., only `weight` requires suspension, as in SMC inference).

To illustrate the semantics, consider $\mathbf{t}_{\text{example}}$ of Fig. 1a again. Because $\mathbf{t}_{\text{example}}$ evaluates precisely one `assume`, the only valid traces for $\mathbf{t}_{\text{example}}$ are singleton traces $[a_1]$, where $a_1 \in \mathbb{R}_{[0,1]}$ due to the Beta prior for a_1. By initially setting ρ to the empty environment $\varnothing$ and following the rules of Fig. 2, we derive $\varnothing \vdash \mathbf{t}_{\text{example}} \; {}^{[a_1]}{\Downarrow}^{f_{\text{Beta}(2,2)}(a_1)\cdot a_1^3(1-a_1)}_{\text{true}} \; a_1$. Note that every evaluation of $\mathbf{t}_{\text{example}}$ has $u = \text{true}$, as there are always four calls to `weight` during evaluation. That is, the derivation requires suspension. However, many subderivations of $\mathbf{t}_{\text{example}}$ do *not* require suspension. For example, the subderivations `assume` (Beta 2 2) and *null obs* do not (i.e., have $u = \text{false}$). Section 4 presents a suspension analysis that conservatively approximates which subderivations require suspension. The analysis enables, e.g., the selective CPS transformation in Fig. 1e.

3.3 A-Normal Form

We simplify the suspension analysis in Section 4 and the selective CPS transformation in Section 5 by requiring that terms are in *A-normal form* (ANF) [13].

Definition 6 (A-normal form). *We define the A-normal form terms* $\mathbf{t}_{ANF} \in T_{ANF}$ *as follows.*

$$\begin{aligned}
\mathbf{t}_{\text{ANF}} &::= x \mid \texttt{let } x = \mathbf{t}'_{\text{ANF}} \texttt{ in } \mathbf{t}_{\text{ANF}} \\
\mathbf{t}'_{\text{ANF}} &::= x \mid c \mid \lambda x.\ \mathbf{t}_{\text{ANF}} \mid x\ y \\
&\quad\ \mid \texttt{if } x \texttt{ then } \mathbf{t}_{\text{ANF}} \texttt{ else } \mathbf{t}_{\text{ANF}} \mid \texttt{assume } x \mid \texttt{weight } x
\end{aligned} \tag{3}$$

It holds that $T_{\text{ANF}} \subset T$. Furthermore, there exist standard transformations to convert terms in T to T_{ANF}. Fig. 3 illustrates Fig. 1a transformed to ANF. We will use Fig. 1a as a running example in Section 4 and Section 5.

Restricting programs to ANF significantly simplifies the suspension analysis and selective CPS transformation. From now on we require that all variable bindings in programs are unique, and together with ANF, the result is that every expression in a program $\mathbf{t} \in T_{\text{ANF}}$ is *uniquely labeled* by a variable name from a `let` expression. This property is essential for the treatment in Section 4.

4 Suspension Analysis

This section presents the main technical contribution: the suspension analysis. The analysis goal is to identify program expressions that may require suspension in the sense of Definition 5. Identifying such expressions leads to the selective CPS transformation in Section 5, enabling transformations such as in Fig 1e.

The suspension analysis builds upon the 0-CFA algorithm [46,39], and we formalize our algorithms based on Lundén et al. [29]. The main challenge we solve is how to model the propagation of suspension in the presence of higher-order functions. The 0 in 0-CFA stands for *context insensitivity*—the analysis considers every part of the program in one global context. Context insensitivity makes the analysis more conservative compared to context-sensitive approaches such as k-CFA, where $k \in \mathbb{N}$ indicates the level of context sensitivity [33]. We use 0-CFA for two reasons: (i) the worst-case time complexity for the analysis is polynomial, while it is exponential for k-CFA already at $k = 1$, and (ii) the limitations of 0-CFA rarely matter in practical PPL applications. For example, k-CFA provides no benefits over 0-CFA for the programs in Section 7.

We assume $\langle \lambda x.\ \mathbf{t}, \rho \rangle \notin C$ (recall that C is the set of intrinsics). That is, we assume that closures are not part of the intrinsics. In particular, this disallows intrinsic operations (including the use of `assume` d, $d \in D \subset C$) to produce closures, which would needlessly complicate the analysis without any benefit.

Consider the program in Fig. 3, and assume that `weight` requires suspension. Clearly, the expression labeled by w_1 at line 20 then requires suspension. Furthermore, w_1 evaluates as part of the larger expression labeled by t_8 at line 10. Consequently, the evaluation of t_8 also requires suspension. Also, t_8 evaluates as part of an application of the abstraction binding *obs* at line 7. In particular, the abstraction binding *obs* binds to *iter*, and we apply *iter* at lines 23 and 33. Thus, the expressions named by t_{17} and t_{22} require suspension. In summary, we have that w_1, t_8, t_{17}, and t_{22} require suspension, and we also note that all applications of the abstraction binding *obs* require suspension.

We proceed to the formalization and first introduce standard *abstract values.*

Definition 7 (Abstract values). *We define the abstract values* $\mathbf{a} \in A$ *as* $\mathbf{a} ::= \lambda x.y \mid \mathtt{const}_x\, n$ *for* $x, y \in X$ *and* $n \in \mathbb{N}$.

The abstract value $\lambda x.y$ represents all closures originating at, e.g., a term $\lambda x.$ `let` y `= 1 in` y in a program at runtime (recall that we assume that the variables x and y are unique). Note that the y indicates the name returned by the body (formalized by the function NAME in Algorithm 1). The abstract value

Algorithm 1 Constraint generation for the suspension analysis. We write the functional-style pseudocode for the algorithm itself in sans serif font to distinguish it from terms in T.

```
function GENERATECONSTRAINTS(t): $T_{\text{ANF}} \to \mathcal{P}(R)$ =
 1 match t with
 2 | $x \to \varnothing$
 3 | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$
 4   GENERATECONSTRAINTS($\mathbf{t}_2$) $\cup$
 5     match $\mathbf{t}_1$ with
 6     | $y \to \{S_y \subseteq S_x\}$
 7     | $c \to$ if $|c| > 0$ then $\{\texttt{const}_x\ |c| \in S_x\}$
 8           else $\varnothing$
 9     | $\lambda y.\ \mathbf{t}_b \to$ GENERATECONSTRAINTS($\mathbf{t}_b$)
10         $\cup\ \{\lambda y.\ $NAME$\ \mathbf{t}_b \in S_x\}$
11         $\cup\ \{suspend_n \Rightarrow suspend_y$
12               $\mid n \in$ SUSPENDNAMES$(t_b)\}$
13     | $lhs\ rhs \to \{$
14         $\forall z \forall y\ \lambda z.y \in S_{lhs}$
15           $\Rightarrow (S_{rhs} \subseteq S_z) \wedge (S_y \subseteq S_x)$,
16         $\forall y \forall n\ \texttt{const}_y\ n \in S_{lhs} \wedge n > 1$
17           $\Rightarrow \texttt{const}_y\ n - 1 \in S_x$,
18         $\forall y\ \lambda y.\_ \in S_{lhs}$
19           $\Rightarrow (suspend_y \Rightarrow suspend_x)$,
20         $\forall y\ \texttt{const}_y\ \_ \in S_{lhs}$
21           $\Rightarrow (suspend_y \Rightarrow suspend_x)$,
22         $suspend_x \Rightarrow$
23           $(\forall y\ \lambda y.\_ \in S_{lhs} \Rightarrow suspend_y)$
24           $\wedge\ (\forall y\ \texttt{const}_y\ \_ \in S_{lhs} \Rightarrow suspend_y)$
25       $\}$
26     | assume _ $\to$
27       if $suspend_{\texttt{assume}}$ then $\{suspend_x\}$ else $\varnothing$
28
29     | weight _ $\to$
30       if $suspend_{\texttt{weight}}$ then $\{suspend_x\}$ else $\varnothing$
31     | if $y$ then $\mathbf{t}_t$ else $\mathbf{t}_e \to$
32       GENERATECONSTRAINTS($\mathbf{t}_t$)
33         $\cup$ GENERATECONSTRAINTS($\mathbf{t}_e$)
34         $\cup\ \{S_{\text{NAME}\ \mathbf{t}_t} \subseteq S_x, S_{\text{NAME}\ \mathbf{t}_e} \subseteq S_x\}$
35         $\cup\ \{suspend_n \Rightarrow suspend_x$
36               $\mid n \in$ SUSPENDNAMES($\mathbf{t}_t$)
37                     $\cup$ SUSPENDNAMES($\mathbf{t}_e$)$\}$
38
39 function NAME(t): $T_{\text{ANF}} \to X$ =
40   match t with
41   | $x \to x$
42   | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$ NAME($\mathbf{t}_2$)
43
44 function SUSPENDNAMES(t): $T_{\text{ANF}} \to \mathcal{P}(X)$ =
45   match t with
46   | $x \to \varnothing$
47   | let $x = \mathbf{t}_1$ in $\mathbf{t}_2 \to$
48     SUSPENDNAMES($\mathbf{t}_2$) $\cup$
49       match $\mathbf{t}_1$ with
50       | $lhs\ rhs \to \{x\}$
51       | if $y$ then $\mathbf{t}_t$ else $\mathbf{t}_e \to \{x\}$
52       | assume _ $\to$
53         if $suspend_{\texttt{assume}}$ then $\{x\}$ else $\varnothing$
54       | weight _ $\to$
55         if $suspend_{\texttt{weight}}$ then $\{x\}$ else $\varnothing$
56       | _ $\to \varnothing$
```

$\texttt{const}_x\ n$ represents all intrinsic functions of arity n originating at x. For example, $\texttt{const}_x\ 2$ originates at, e.g., a term `let` x `=` $+$ `in` **t**.

The central objects in the analysis are sets $S_x \in \mathcal{P}(A)$ and boolean values $suspend_x$ for all $x \in X$. The set S_x contains all abstract values that may flow to the expression labeled by x, and $suspend_x$ indicates whether or not the expression requires suspension. A trivial but useless solution is $S_x = A$ and $suspend_x = \text{true}$ for all variables x in the program. To get more precise information regarding suspension, we wish to find smaller solutions to the S_x and $suspend_x$.

To formalize the set of sound solutions for S_x and $suspend_x$, we generate *constraints* $\mathbf{c} \in R$ for programs.[†] Algorithm 1 formalizes the necessary constraints for programs $\mathbf{t} \in T_{\text{ANF}}$ with a function GENERATECONSTRAINTS that recursively traverses the program **t** to generate a set of constraints. Due to ANF, there are only two cases in the top match (line 1). Variables generate no constraints, and the important case is for `let` expressions at lines 3–30. The algorithm makes use of an auxiliary function NAME (line 39) that determines the name of an ANF expression, and a function SUSPENDNAMES (line 44) that determines the names of all top-level expressions within an expression that may suspend (namely, applications, `if` expressions, and `assume` and/or `weight`).

We next illustrate and motivate the generated constraints by considering the set of constraints $\textsc{generateConstraints}(\mathbf{t}_{\text{example}})$, where $\mathbf{t}_{\text{example}}$ is the program in Fig. 3. Many constraints are standard, and we therefore focus on the new suspension constraints introduced as part of this paper. In particular, the challenge is to correctly capture the flow of suspension requirements across function applications and higher-order functions. First, we see that defining aliases (line 6) generates constraints of the form $S_y \subseteq S_x$, that constants introduce `const` abstract values (e.g., $\texttt{const}_{t_6} 1 \in S_{t_6}$), and that `assume` and `weight` introduce suspension requirements, e.g., $\mathit{suspend}_{w_1}$ (shorthand for $\mathit{suspend}_{w_1} = \text{true}$).

First, we consider the constraints generated for $\lambda \mathit{obs}.$ (line 7 in Fig. 3) through the case at lines 9-12 in Algorithm 1. To keep the example simple, we treat the unexpanded `let rec` as an ordinary `let` in the analysis (for this particular example, the analysis result is unaffected). Omitting the recursively generated constraints for the abstraction body, the generated constraints are

$$\{\lambda \mathit{obs}.\, t_8 \in S_{\mathit{iter}}\} \cup \{\mathit{suspend}_n \Rightarrow \mathit{suspend}_{\mathit{obs}} \mid n \in \{t_7, t_8\}\}. \tag{4}$$

The first constraint is standard and states that the abstract value $\lambda \mathit{obs}.\, t_8$ flows to S_{iter} as the variable naming the $\lambda \mathit{obs}$ expression is t_8 at line 26 in Fig. 3 (difficult to notice due to the column breaks). The remaining constraints are new and sets up the flow of suspension requirements. Specifically, the abstraction *obs* itself requires suspension if any expression bound by a top-level `let` in its body requires suspension. For efficiency, we only set up dependencies for expressions that may suspend (formalized by $\textsc{suspendNames}$ in Algorithm 1). Note here that we do not add the constraint $\mathit{suspend}_{w_1} \Rightarrow \mathit{suspend}_{\mathit{obs}}$, as w_1 is not at top-level in the body of *obs*. Instead, we later add the constraint $\mathit{suspend}_{w_1} \Rightarrow \mathit{suspend}_{t_8}$, and $\mathit{suspend}_{w_1} \Rightarrow \mathit{suspend}_{\mathit{obs}}$ follows by transitivity.

The constraints generated for the `if` bound to t_8 at line 10 through the case at lines 31-37 in Algorithm 1 are (omitting recursively generated constraints)

$$\begin{aligned}&\{S_{t_9} \subseteq S_{t_8}, S_{t_{17}} \subseteq S_{t_8}\}\\ &\qquad \cup \{\mathit{suspend}_n \Rightarrow \mathit{suspend}_{t_8} \mid n \in \{t_{11}, t_{13}, t_{14}, w_1, t_{16}, t_{17}\}\}.\end{aligned} \tag{5}$$

The first two constraints are standard, and state that abstract values in the results of both branches flow to the result S_{t_8}. The last set of constraints is new and similar to the abstraction suspension constraints. The constraints capture that all expressions at top-level in both branches that require suspension also cause t_8 to require suspension.

Consider the application at line 23 in Fig. 3. The generated constraints through the case at lines 13-25 in Algorithm 1 are

$$\begin{aligned}\{\ &\forall z \forall y\ \ \lambda z.y \in S_{\mathit{iter}} \Rightarrow (S_{t_{16}} \subseteq S_z) \wedge (S_y \subseteq S_{t_{17}}),\\ &\forall y \forall n\ \ \texttt{const}_y\, n \in S_{\mathit{iter}} \wedge n > 1 \Rightarrow \texttt{const}_y\, n-1 \in S_{t_{17}},\\ &\forall y\ \ \lambda y.\_ \in S_{\mathit{iter}} \Rightarrow (\mathit{suspend}_y \Rightarrow \mathit{suspend}_{t_{17}}),\\ &\forall y\ \ \texttt{const}_y\ \ \_ \in S_{\mathit{iter}} \Rightarrow (\mathit{suspend}_y \Rightarrow \mathit{suspend}_{t_{17}}),\\ &\mathit{suspend}_{t_{17}} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{\mathit{iter}} \Rightarrow \mathit{suspend}_y)\\ &\qquad\qquad\qquad \wedge (\forall y\ \ \texttt{const}_y\ \ \_ \in S_{\mathit{iter}} \Rightarrow \mathit{suspend}_y)\ \}.\end{aligned} \tag{6}$$

The first two constraints are standard and state how abstract values flow as a result of applications. The last three constraints are new and relate to suspension. The third and fourth constraints state that if an abstraction or intrinsic requiring suspension flows to *iter*, the result t_{17} of the application also requires suspension. The fifth constraint states that if the result t_{17} requires suspension, then *all* abstractions and constants flowing to *iter* require suspension. This last constraint is not strictly required to later prove the soundness of the analysis in Theorem 1, but, as we will see in Section 5, it is required for the selective CPS transformation.

We find a solution to the constraints through a fairly standard algorithm that propagates abstract values according to the constraints until fixpoint.[†] However, we extend the algorithm to support the new suspension constraints. The algorithm is a function $\textsc{analyzeSuspend}\colon T_{\text{ANF}} \to ((X \to \mathcal{P}(A)) \times \mathcal{P}(X))$. The function returns a map $\mathsf{data} : X \to \mathcal{P}(A)$ that assigns sets of abstract values to all S_x and a set $\mathsf{suspend} : \mathcal{P}(X)$ that assigns $\mathit{suspend}_x = \text{true}$ iff $x \in \mathsf{suspend}$. Importantly, the assignments to S_x and $\mathsf{suspend}_x$ satisfy all generated constraints. To illustrate the algorithm, here are the analysis results $\textsc{analyzeSuspend}(\mathbf{t}_{\text{example}})$:

$$\begin{gathered}
S_{iter} = \{\lambda obs.t_8\} \quad S_{t_6} = \{\texttt{const}_{t_6} 1\} \quad S_{t_{10}} = \{\texttt{const}_{t_{10}} 2\} \\
S_{t_{11}} = \{\texttt{const}_{t_{10}} 1\} \quad S_{t_{12}} = \{\texttt{const}_{t_{12}} 1\} \quad S_{t_{15}} = \{\texttt{const}_{t_{15}} 1\} \\
S_n = \varnothing \mid \text{all other } n \in X \\
\mathit{suspend}_n = \text{true} \mid n \in \{obs, w_1, t_8, t_{17}, t_{22}\} \\
\mathit{suspend}_n = \text{false} \mid \text{all other } n \in X.
\end{gathered} \tag{7}$$

The above results confirm our earlier reasoning: the expressions labeled by *obs*, w_1, t_8, t_{17}, and t_{22} may require suspension.

We now consider the soundness of the analysis. First, the soundness of 0-CFA is well established (see, e.g., Nielson et al. [39]) and extends to our new constraints, and we take the following lemma to hold without proof.

Lemma 1 (0-CFA soundness). *For every* $\mathbf{t} \in T_{ANF}$*, the solution given by* $\textsc{analyzeSuspend}(\mathbf{t})$ *for* S_x *and* $\mathit{suspend}_x$*,* $x \in X$*, satisfies the constraints* $\textsc{generateConstraints}(\mathbf{t})$*.*

Next, we must show that the constraints themselves are sound. Consider the evaluation of an arbitrary term $\mathbf{t} \in T_{\text{ANF}}$. For each subderivation of $\mathbf{t}$, labeled by a name x (due to ANF), it must hold that $\mathit{suspend}_x = \text{true}$ if the subderivation requires suspension. Otherwise, the analysis is unsound. Theorem 1 formally captures the soundness. Note that the analysis is conservative (i.e., incomplete), because it may find $\mathit{suspend}_x = \mathit{true}$ even if the subderivation for x does not require suspension.

Theorem 1 (Suspension analysis soundness). *Let* $\mathbf{t} \in T_{ANF}$*,* $s \in S$*,* $u \in \{\mathit{false}, \mathit{true}\}$*,* $w \in \mathbb{R}$*, and* $\mathbf{v} \in V$ *such that* $\varnothing \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u}\; \mathbf{v}$*. Now, let* S_x *and* $\mathit{suspend}_x$ *for* $x \in X$ *according to* $\textsc{analyzeSuspend}(\mathbf{t})$*. For every subderivation* $(\rho \vdash \texttt{let}\ x = \mathbf{t}_1\ \texttt{in}\ \mathbf{t}_2 \;{}^{s_1 \| s_2}\!\Downarrow^{w_1 \cdot w_2}_{u_1 \vee u_2}\; \mathbf{v}')$ *of* $(\varnothing \vdash \mathbf{t} \;{}^{s}\!\Downarrow^{w}_{u}\; \mathbf{v})$*,* $u_1 = \mathit{true}$ *implies* $\mathit{suspend}_x = \mathit{true}$*.*

Algorithm 2 Selective continuation-passing style transformation. We define $\mathbf{t}_{\text{id}} = \lambda x.x$. The term c_{CPS} is the CPS version of c. We write the functional-style pseudocode for the algorithm itself in sans serif font to distinguish it from terms in T.

```
function CPS(vars, t): $\mathcal{P}(X) \times T_{\text{ANF}} \to T^+$ =
return CPS'($\mathbf{t}_{\text{id}}$, t)

function CPS'(cont,t): $T \times T_{\text{ANF}} \to T^+$ =
  match t with
  | $x$ → if cont = $\mathbf{t}_{\text{id}}$ then t else cont t
  | let $x = \mathbf{t}_1$ in $\mathbf{t}_2$ →
    let $\mathbf{t}'_2$ = CPS'(cont, $\mathbf{t}_2$) in
    match $\mathbf{t}_1$ with
    | $y$ → let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$
    | $c$ → let $x$ =
              (if $x \in$ vars then $c_{\text{CPS}}$ else $c$) in $\mathbf{t}'_2$
    | $\lambda y.\ \mathbf{t}_b$ →
      let $\mathbf{t}'_1$ = if $y \in$ vars
        then $\lambda k.\lambda y.$ CPS'($k, \mathbf{t}_b$)
        else $\lambda y.$ CPS'($\mathbf{t}_{\text{id}}, \mathbf{t}_b$)
      in
      let $x = \mathbf{t}'_1$ in $\mathbf{t}'_2$
    | $lhs\ rhs$ →
      if $x \in$ vars then
        if TAILCALL(t)
        then $lhs$ cont $rhs$
        else $lhs$ ($\lambda x.\mathbf{t}'_2$) $rhs$
      else let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$
    | if $y$ then $\mathbf{t}_t$ else $\mathbf{t}_e$ →
      if $x \in$ vars then
        if TAILCALL(t) then
          if $y$ then CPS'(cont, $\mathbf{t}_t$)
          else CPS'(cont, $\mathbf{t}_e$)
        else
          let $k = \lambda x.\mathbf{t}'_2$ in
          if $y$ then CPS'($k, \mathbf{t}_t$) else CPS'($k, \mathbf{t}_e$)
      else let $x$ = if $y$ then CPS'($\mathbf{t}_{\text{id}}, \mathbf{t}_t$)
                   else CPS'($\mathbf{t}_{\text{id}}, \mathbf{t}_e$) in $\mathbf{t}'_2$
    | assume $y$ → let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$
      if $x \in$ vars then
        if TAILCALL(t)
        then Suspension$_{\text{assume}}$($y$, cont)
        else Suspension$_{\text{assume}}$($y$,$\lambda x.$CPS'(cont, $\mathbf{t}_2$))
      else let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$
    | weight $y$ → let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$
      if $x \in$ vars then
        if TAILCALL(t)
        then Suspension$_{\text{weight}}$($y$, cont)
        else Suspension$_{\text{weight}}$($y$,$\lambda x.$CPS'(cont, $\mathbf{t}_2$))
      else let $x = \mathbf{t}_1$ in $\mathbf{t}'_2$

function TAILCALL(t): $T_{\text{ANF}} \to \{\text{false}, \text{true}\}$ =
  match t with
  | let $x$ = _ in $x$ → true
  | _ → false
```

The proof uses Lemma 1 and structural induction over the derivation $\varnothing \vdash \mathbf{t}\ {}^{s}\!\Downarrow^{w}_{u}\ \mathbf{v}$.[†]

Next, we use the suspension analysis to selectively CPS transform programs.

5 Selective CPS Transformation

This section presents the second technical contribution: the selective CPS transformation. The transformations themselves are standard, and the challenge is to correctly use the suspension analysis results for a selective transformation.

Algorithm 2 is the full algorithm. Using terms in ANF as input significantly helps reduce the algorithm's complexity. The main function CPS takes as input a set vars : $\mathcal{P}(X)$, indicating which expressions to CPS transform, and a program $\mathbf{t} \in T_{\text{ANF}}$ to transform. It is the new vars argument that separates the transformation from a standard CPS transformation. For the purposes of this paper, we always use vars $= \{x \mid suspend_x = \text{true}\}$, where the $suspend_x$ come from ANALYZESUSPEND(**t**). One could also use vars $= X$ for a standard full CPS transformation (e.g., Fig 1f), or some other set vars for other application domains. The value returned from the CPS function is a (non-ANF) term of the

```
let t1 = 2 in
let t2 = 2 in
let t3 = Beta in
let t4 = t3 t1 in
let t5 = t4 t2 in
let a1 = assume t5 in
let rec iter = λk. λobs.
  let t6 = null in
  let t7 = t6 obs in
  if t7 then
    let t9 = () in
    t9
  else
    let t10 = f_Bernoulli in
    let t11 = t10 a1 in
    let t12 = head in
    let t13 = t12 obs in
    let t14 = t11 t13 in
    Suspension_weight(t14,
      λ_.
        let t15 = tail in
        let t16 = t15 obs in
        iter k t16)
in
let t18 = true in
let t19 = false in
let t20 = true in
let t21 = true in
let t22 = [t21,t20,t19,t18] in
let k' = λ_. a1 in
iter k' t22
```

Fig. 4: The running example from Fig. 3 after selective CPS transformation. The program is semantically equivalent to Fig. 1e.

type T^+. The helper function CPS′, initially called at line 1, takes as input a continuation term cont, indicating the continuation to apply in tail position. Initially, this continuation term is $\mathbf{t}_{\text{id}}$, which indicates no continuation. Similarly to Algorithm 1, the top-level match at line 4 has two cases: a simple case for variables (line 5) and a complex case for `let` expressions (lines 6–49). To enable optimization of tail calls, the auxiliary function TAILCALL indicates whether or not an ANF expression is a tail call (i.e., of the form `let` x = $\mathbf{t}'$ `in` x).

We now illustrate Algorithm 2 by computing CPS($\mathsf{vars}_{\text{example}}, \mathbf{t}_{\text{example}}$), where $\mathsf{vars}_{\text{example}} = \{obs, w_1, t_8, t_{17}, t_{22}\}$ is from (7), and $\mathbf{t}_{\text{example}}$ is from Fig. 3. Fig. 4 presents the final result. First, we note that the transformation does not change expressions not labeled by a name in $\mathsf{vars}_{\text{example}}$, as they do not require suspension. In the following, we therefore focus only on the transformed expressions. First, consider the abstraction *obs* defined at line 7 in Fig. 3, handled by the case at line 12 in Algorithm 2. As $obs \in \mathsf{vars}_{\text{example}}$, we apply the standard CPS transformation for abstractions: add a continuation parameter to the abstraction and recursively transform the body with this continuation. Next, consider the transformation of the `weight` expression w_1 at line 20 in Fig. 3, handled by the case at line 44 in Algorithm 2. The expression is not at tail position, so we build a new continuation containing the subsequent `let` expressions, recursively transform the body of the continuation, and then wrap the end result in a Suspension object. The `if` expression t_8 at line 10 in Fig. 3, handled by the case at line 28 in Algorithm 2, is in tail position (it is directly followed by returning t_8). Consequently, we transform both branches recursively. Finally, we have the applications t_{17} and t_{22} at lines 23 and 33 in Fig. 3, handled by the case at line 18 in Algorithm 2. The application t_{17} is at tail position, and we transform it by adding the current continuation as an argument. The application at t_{22} is not at tail position, so we construct a continuation k' that returns the final value a_1 (line 34 in Fig. 3), and then add it as an argument to the application.

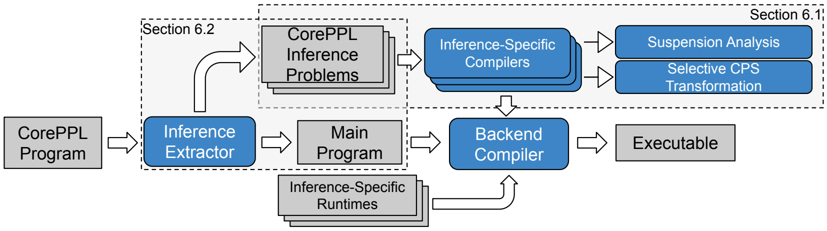


Fig. 5: Overview of the Miking CorePPL compiler implementation. We divide the overall compiler into two parts, (i) *suspension analysis and selective CPS* (Section 6.1) and (ii) *inference problem extraction* (Section 6.2). The figure depicts artifacts as gray rectangular boxes and transformation units and libraries as blue rounded boxes. Note how the *inference extractors* transformation separates the program into two different paths that are combined again after the inference-specific compilation. The white inheritance arrows (pointing to *suspension analysis* and *selective CPS transformations*) mean that these libraries are used within the inference-specific compiler transformation.

It is not guaranteed that Algorithm 2 produces a correct result. Specifically, for all applications *lhs rhs*, we must ensure that (i) if we CPS transform the application, we must also CPS transform *all* possible abstractions that can occur at *lhs*, and (ii) if we do *not* CPS transform the application, we must *not* CPS transform any abstraction that can occur at *lhs*. We control this through the argument vars. In particular, assigning vars according to the suspension analysis produces a correct result. To see this, consider the application constraints at lines 13–25 in Algorithm 1 again, and note that if any abstraction or intrinsic operation that requires suspension occur at *lhs*, $suspend_x = \text{true}$. Furthermore, the last application constraint ensures that if $suspend_x = \text{true}$, then *all* abstractions and intrinsic operations that occur at *lhs* require suspension. Consequently, for all $\lambda y.\ \_$ and $const_y\ \_$, either all $suspend_y = \text{true}$ or all $suspend_y = \text{false}$.

6 Implementation

We implement the suspension analysis and selective CPS transformation in Miking CorePPL [30], a core PPL implemented in the domain-specific language construction framework Miking [9]. We choose Miking CorePPL for the implementation over other CPS-based PPLs, as the language implementation contains an existing 0-CFA base implementation which simplifies the suspension analysis implementation. Fig. 5 presents the organization of the CorePPL compiler. The input is a CorePPL program that may contain many inference problems and applications of inference algorithms, similar to WebPPL and Anglican. The output is an executable produced by one of the Miking backend compilers. Section 6.1 gives the details of the suspension analysis and selective CPS implementations, and in particular the differences compared to the core calculus in Section 3. Section 6.2 presents the inference extractor and its operation combined with selective CPS. The suspension analysis, selective CPS transformation, and inference extraction implementations consist of roughly 1500 lines of code (a contribution in this paper). The code is available on GitHub [2].

6.1 Suspension Analysis and Selective CPS

Miking CorePPL extends the abstract syntax in Definition 1 with standard functional data structures and features such as algebraic data types (records, tuples, and variants), lists, and pattern matching. The suspension analysis and selective CPS implementations in Miking CorePPL extend Algorithm 1 and Algorithm 2 to support these language features. Furthermore, compared to $suspend_{\texttt{weight}}$ and $suspend_{\texttt{assume}}$ in Fig. 2, the implementation allows arbitrary configuration of suspension sources. In particular, the implementation uses this arbitrary configuration together with the alignment analysis by Lundén et al. [29]. This combination allows selectively CPS transforming to suspend at a subset of `assume`s or `weight`s for aligned versions of SMC and MCMC inference algorithms.

Miking CorePPL also includes a framework for inference algorithm implementation. Specifically, to implement new inference algorithms, users implement an *inference-specific compiler* and *inference-specific runtime*. Fig. 5 illustrates the different compilers and runtimes. Each inference-specific compiler applies the suspension analysis and selective CPS transformation to suit the inference algorithm's particular suspension requirements.

Next, we show how Miking CorePPL handles programs containing many inference problems solved with different inference algorithms.

6.2 Inference Problem Extraction

Fig. 5 includes the inference extraction compiler procedure. First, the compiler applies an inference extractor to the input program. The result is a set of inference problems and a main program containing remaining glue code. Second, the compiler applies inference-specific compilers to each inference problem. Finally, the compiler combines the main program and the compiled inference problems with inference-specific runtimes and supplies the result to a backend compiler.

Consider the example in Fig. 6a. We define a function `m` that constructs a minimal inference problem on lines 7–10, using a single call to `assume` and a single call to `observe` (modifying the execution weight similar to `weight`). The function takes an initial probability distribution `d` and a data point `y` as input. We apply aligned lightweight MCMC inference for the inference problem through the `infer` construct on lines 12–16. The first argument to `infer` gives the inference algorithm configuration, and the second argument the inference problem. Inference problems are thunks (i.e., functions with a dummy unit argument). We construct the inference problem thunk by an application of `m` with a uniform initial distribution and data point 1.0. The inference result `d0` is another probability distribution, and we use it as the first initial distribution in the recursive

```
 1 mexpr
 2 let data = [
 3   24.0, 42.2, 96.7, 9.2, 85.8,
 4   34.2, 41.7, 53.4, 85.6, 45.4
 5 ] in
 6
 7 let m = lam d. lam y. lam.
 8  let x = assume d in
 9  observe y (Gaussian x 0.1);
10  x in
11
12 let d0 =
13  infer (LightweightMCMC
14   { iterations = 100,
15     aligned = true })
16   (m (Uniform 0.0 4.0) 1.0) in
17
18 recursive let repeat =
19  lam data. lam d.
20   match data with [y] ++ data then
21    let posterior =
22     infer (BPF {particles = 100})
23            (m d y) in
24    repeat data posterior
25   else d
26 let d1 = repeat data d0 in
27 match distEmpiricalSamples d1
28 with (samples, weights) in
29 iter
30  (lam s.
31    print
32     (concat (float2string s) "\n"))
33  samples
```

(a) Miking CorePPL program.

```
1 let m = lam d. lam y. lam.
2   let x = assume d in
3   observe y (Gaussian x 0.1);
4   x in
5 m (Uniform 0.0 4.0) 1.0 ()
```

(b) Extracted inference problem from line 13 in (a).

```
1 let m = lam d. lam y. lam.
2   let x = assume d in
3   observe y (Gaussian x 0.1);
4   x in
5 m d y ()
```

(c) Extracted inference problem from line 22 in (a).

Fig. 6: Example Miking CorePPL program in (a) with two non-trivial uses of `infer`. Figures (b) and (c) show the extracted and selectively CPS-transformed inference problems at lines 13 and 22 in (a), respectively. The compiler handles the free variables `d` and `y` in (c) in a later stage.

`repeat` function (lines 19–24). This function repeatedly performs inference using the SMC bootstrap particle filter (lines 21–23), again using the function `m` to construct the sequence of inference problems. Each `infer` application uses the result distribution from the previous iteration as the initial distribution and consumes data points from the `data` sequence. We extract and print the samples from the final result distribution `d1` at lines 29–33. A limitation with the current extraction approach is that we do not yet support nested `infer`s.

A key challenge in the compiler design is how to handle different inference algorithms within one probabilistic program. In particular, inference algorithms require different selective CPS transformations, applied to different parts of the code. To allow the separate handling of inference algorithms, we apply the extraction approach by Hummelgren et al. [22] on the `infer` applications, producing separate inference problems for each occurrence of `infer`. Although the compiler design mostly concerns rather comprehensive engineering work, special care must be taken to handle the non-trivial problem of name bindings when transforming and combining different code entities together. For instance, the compiler must selectively CPS transform Fig. 6b to suspend at `assume` (required by MCMC) and selectively CPS transform Fig. 6c to suspend at `observe` (required by SMC). We design a robust and modular solution, where it is possible to easily add new inference algorithms without worrying about name conflicts.

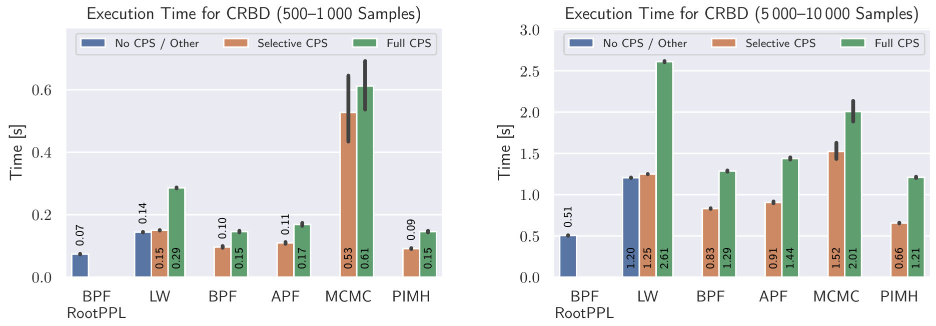
7 Evaluation

This section presents the evaluation of the suspension analysis and selective CPS implementations. Our main claims are that (i) the approach of selective CPS significantly improves performance compared to traditional full CPS, and (ii) that this holds for a significant set of inference algorithms, evaluated on realistic inference problems. We use four PPL models and corresponding data sets from the Miking benchmarks repository, available on GitHub [1]. The models are: constant rate birth-death (CRBD) in Section 7.1, cladogenetic diversification rate shift (ClaDS) in Section 7.2, latent Dirichlet allocation (LDA) in Section 7.3, and vector-borne disease (VBD) in Section 7.4. All models are significant and actively used in different research areas: CRBD and ClaDS in evolutionary biology and phylogenetics [37,43,32], LDA in topic modeling [7], and VBD in epidemiology [14,34]. In addition to the Miking CorePPL models from the Miking benchmarks, we also implement CRBD in WebPPL and Anglican.

We add a number of popular inference algorithms in Miking CorePPL with support for selective CPS. The first is standard likelihood weighting (LW), as introduced in Section 2. LW does not strictly require CPS, but we implement it with suspensions at `weight` to highlight the difference between no CPS, selective CPS, and full CPS. LW gives a good direct measure of CPS overhead as the algorithm simply executes programs many times. Suspending at `weight` can also be useful in LW to stop executions with weight 0 (i.e., useless samples) early. However, we do not use early stopping to isolate the effect CPS has on execution time. Next, we add the bootstrap particle filter (BPF) and alive particle filter (APF). Both are SMC algorithms that suspend at `weight` to resample executions. BPF is a standard algorithm often used in PPLs, and APF is a related algorithm introduced in a PPL context by Kudlicka et al. [24]. The final two inference algorithms we add are aligned lightweight MCMC (just MCMC for short) and particle-independent Metropolis–Hastings. Aligned lightweight MCMC [29] is an extension to the standard PPL Metropolis–Hastings approach introduced by Wingate et al. [49], and suspends at a subset of calls to `assume`. Particle-independent Metropolis–Hastings (PIMH) is an MCMC algorithm that repeatedly uses the BPF (suspending at `weight`) within a Metropolis–Hastings MCMC algorithm [40]. We limit the scope to single-core CPU inference.

In addition to the inference algorithms in Miking CorePPL, we also use three other state-of-the-art PPLs for CRBD: Anglican, WebPPL, and the special high-performance RootPPL compiler for Miking CorePPL [30]. For Anglican, we apply LW, BPF, and PIMH inference. For WebPPL, we use BPF and (non-aligned) lightweight MCMC. For the RootPPL version of Miking CorePPL, we use BPF inference (the only supported inference algorithm).

We consider two configurations for each model: 1 000 and 10 000 samples. An exception is for CRBD and ClaDS, where we adjust APF to use 500 and 5 000



	Anglican LW	Anglican BPF	WebPPL BPF	WebPPL MCMC
1 000 samples	11.6 ± 0.36 s	5.65 ± 2.71 s	2.42 ± 0.20 s	1.42 ± 0.07 s
10 000 samples	90.4 ± 2.12 s	29.1 ± 2.35 s	53.9 ± 4.03 s	3.10 ± 0.77 s

Fig. 7: Mean execution times for the CRBD model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`). The table shows standard deviations.

samples to make the inference accuracy comparable to the related BPF. We run each experiment 300 times (with one warmup run) and measure execution time (excluding compile time). To justify the efficiency of the suspension analysis and selective CPS transformation that are part of the compiler, we note here that they, combined, run in only 1–5 ms for all models.

The experiments do *not* compare the performance of different inference algorithms. To do this, one would also need to consider how accurate the inference results are for a given amount of execution time. Accuracy varies dramatically between different combinations of inference algorithms and models. We evaluate the execution time of selective and full CPS in isolation for individual inference algorithms. Selective CPS is solely an execution time optimization—the algorithms themselves and their accuracy remain unchanged.†

For Miking CorePPL, we used OCaml 4.12.0 as backend compiler for the implementation in Section 6 and GCC 7.5.0 for the separate RootPPL compiler. We used Anglican 1.1.0 (OpenJDK 11.0.19) and WebPPL 0.9.15 (Node.js 16.18.0). We ran the experiments on an Intel Xeon Gold 6148 CPU with 64 GB of memory using Ubuntu 18.04.6.

7.1 Constant Rate Birth-Death

CRBD is a diversification model, used by evolutionary biologists to infer distributions over birth and death rates for observed evolutionary trees of groups of species, called *phylogenies*. For the CRBD experiment, we use the Alcedinidae phylogeny (Kingfisher birds, 54 extant species) [43,23]. We compare CRBD in Miking CorePPL (55 lines of code)†, Anglican (129 lines of code)†, and WebPPL (66 lines of code)†. The total experiment execution time was 9 hours.

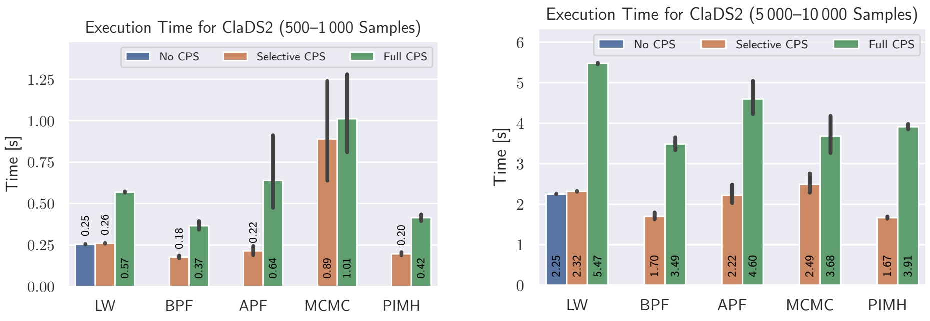


Fig. 8: Mean execution times for the ClaDS model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

Fig. 7 presents the results. We note that selective CPS is faster than full CPS in all cases. Unlike full CPS, the overhead of selective CPS compared to no CPS is marginal for LW. The execution time for early MCMC samples is sensitive to initial conditions, and we therefore see more variance for MCMC compared to the other algorithms. When we increase the number of samples to 10 000, the variance reduces. With the exception of MCMC in WebPPL, the execution times for Anglican and WebPPL are one order of magnitude slower than the equivalent algorithms in Miking CorePPL. However, note that the comparison is only for reference and not entirely fair, as Anglican and WebPPL use different execution environments compared to Miking CorePPL. Lastly, we note that the Miking CorePPL BPF implementation with selective CPS is not much slower than when compiling Miking CorePPL to RootPPL BPF—a compiler designed specifically for efficiency (but with other limitations, such as the lack of garbage collection). RootPPL does not use CPS, and instead enables suspension through a low-level transformation using the concept of PPL control-flow graphs [30].

7.2 Cladogenetic Diversification Rate Shift

ClaDS is another diversification model used in evolutionary biology [32,43]. Unlike CRBD, it allows birth and death rates to change over time. We again use the Alcedinidae phylogeny. The source code consists of 72 lines of code.[†] The total experiment execution time was 3 hours. Fig. 8 presents the results. We note that selective CPS is faster than full CPS in all cases.

7.3 Latent Dirichlet Allocation

LDA [7] is a model from natural language processing used to categorize documents into *topics*. We use a synthetic data set with size comparable to the data set in Ritchie et al. [41]: a vocabulary of 100 words, 10 topics, and 25 observed documents (30 words in each). We do not apply any optimization techniques such as collapsed Gibbs sampling [21]. Solving the inference problem using a PPL is

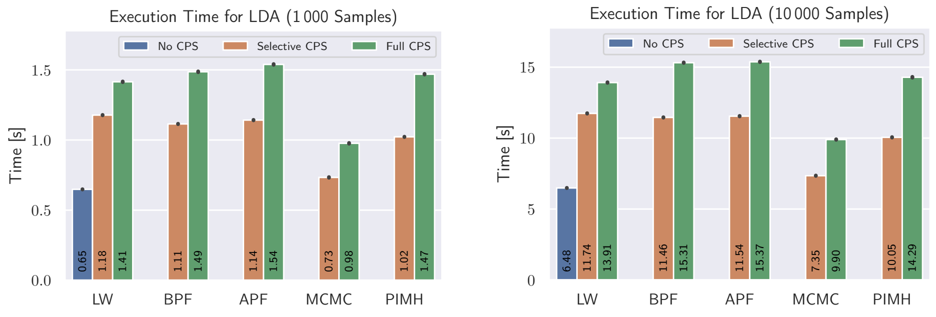


Fig. 9: Mean execution times for the LDA model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

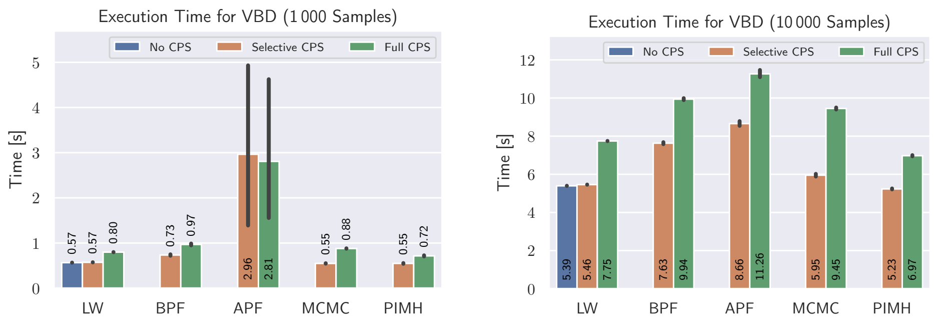


Fig. 10: Mean execution times for the VBD model. The error bars show 95% confidence intervals (using the option `('ci', 95)` in Seaborn's `barplot`).

therefore challenging already for small data sets. The source code consists of 26 lines of code.[†] The total experiment execution time was 12 hours.

Fig. 9 presents the results. We note that selective CPS is faster than full CPS in all cases. Interestingly, the reduction in overhead compared to full CPS for LW is not as significant. The reason is that suspension at `weight` for the model requires that we CPS transform the most computationally expensive recursion.

7.4 Vector-Borne Disease

We use the VBD model from Funk et al. [14] and later Murray et al. [34]. The background is a dengue outbreak in Micronesia and the spread of disease between mosquitos and humans. The inference problem is to find the true numbers of susceptible, exposed, infectious, and recovered (SEIR) individuals each day, given daily reported numbers of new cases at health centers. The source code consists of 140 lines of code.[†] The total execution time was 8 hours.

Fig. 10 presents the results. Again, we note that selective CPS is faster than full CPS in all cases, except seemingly for APF and 1 000 samples. This is very likely a statistical anomaly, as the variance for APF is quite severe for the case with 1 000 samples. Compared to the BPF, APF uses a resampling approach for which the execution time varies a lot if the number of samples is too low [24]. The plots clearly show this as, compared to 1 000 samples, the variance is reduced to BPF-comparable levels for 10 000 samples. In summary, the evaluation demonstrates the clear benefits of selective CPS over full CPS for universal PPLs.

8 Related Work

There are a number of universal PPLs that require non-trivial suspension. One such language is Anglican [50], which solves the suspension problem using CPS. Anglican performs a full CPS transformation with one exception—certain statically known functions named *primitive procedures*, that include a subset of the regular Clojure (the host language of Anglican) functions, are guaranteed to not execute PPL code, and Anglican does not CPS transform them [47]. However, higher-order functions in Clojure libraries cannot be primitive procedures, and Anglican must manually reimplement such functions (e.g., `map` and `fold`). Anglican does not consider a selective CPS transformation of PPL code, and always fully CPS transforms the PPL part of Anglican programs.

WebPPL [18] and the approach by Ritchie et al. [41] also make use of CPS transformations to implement PPL inference. They do not, however, consider selective CPS transformations. Ŝcibior et al. [44] present an architectural design for a probabilistic functional programming library based on monads and monad transformers (and corresponding theory in Ŝcibior et al. [45]). In particular, they use a coroutine monad transformer to suspend SMC inference. This approach is similar to ours in that it makes use of high-level functional language features to enable suspension. They do not, however, consider a selective transformation.

The PPLs Pyro [6], Stan [10,5], Gen [11,27], and Edward [48] either implement inference algorithms that do not require suspension (e.g., Hamiltonian Monte Carlo), or restrict the language in such a way that suspension is explicit and trivially handled by the language implementation. For example, SMC in Pyro[8] and newer versions of Birch require that users explicitly write programs as a `step` function that the SMC implementation calls iteratively. Resampling only occurs in between calls to `step`, and suspension is therefore trivial.

Work on general-purpose selective CPS transformations include Nielsen [38], Asai and Uehara [4], Rompf et al. [42], and Leijen [26]. They consider typed languages, unlike the untyped language in this paper. The early work by Nielsen [38] considers the efficient implementation of `call/cc` through a selective CPS transformation. The transformation requires manual user annotations, unlike the fully automatic approach in this paper. A more recent approach is due to Asai and Uehara [4], who consider an efficient implementation of delimited continuations using `shift` and `reset` through a selective CPS transformation. Similar to us, they automatically determine where to selectively CPS transform programs. They use an approach based on type inference, while our approach builds upon 0-CFA. Rompf et al. [42] follow a similar approach to Asai and Uehara [4], but for

[8] Note that the main inference algorithm in Pyro is stochastic variational inference, which does not require suspension.

Scala, and additionally require user annotations. Leijen [26] uses a type-directed selective CPS transformation to compile algebraic effect handlers.

There are low-level alternatives to CPS for suspension in PPLs. In particular, there are various languages and approaches that directly implement support for non-preemptive multitasking (e.g., coroutines). Turing [15] and older versions of Birch [36,35] implement coroutines to enable arbitrary suspension, but do not discuss the implementations in detail. Lundén et al. [30] introduces and uses the concept of PPL control-flow graphs to compile Miking CorePPL to the low-level C++ framework RootPPL. The compiler explicitly introduces code that maintains special execution call stacks, distinct from the implicit C++ call stacks. The implementation results in excellent performance, but supports neither garbage collection nor higher-order functions. Another low-level approach is due to Paige and Wood [40], who exploits mutual exclusion locks and the `fork` system call to suspend and resample SMC executions. In theory, many of the above low-level alternatives to CPS can, if implemented efficiently, result in the least possible overhead due to more fine-grained low-level control. The approaches do, however, require significantly more implementation effort compared to a CPS transformation. Comparatively, the selective CPS transformation is a surprisingly simple, high-level, and easy-to-implement alternative that brings the overhead of CPS closer to that of more low-level approaches.

9 Conclusion

This paper introduces a selective CPS transformation for the purpose of execution suspension in PPLs. To enable the transformation, we develop a static suspension analysis that determines parts of programs that require a CPS transformation as a consequence of inference algorithm suspension requirements. We implement the suspension analysis, selective CPS transformation, and an inference problem extraction procedure (required as a result of the selective CPS transformation) in Miking CorePPL. Furthermore, we evaluate the implementation on real-world models from phylogenetics, topic-modeling, and epidemiology. The results demonstrate significant speedups compared to the standard full CPS suspension approach for a large number of Monte Carlo inference algorithms.

Acknowledgments. This project was financially supported by the Swedish Foundation for Strategic Research (FFL15-0032 and RIT15-0012), partially supported by the Swedish Research Council (Grant No. 2018-04329), and by Digital Futures (the DLL project). The research has also been carried out as part of the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications at KTH Royal Institute of Technology. We thank Gizem Çaylak for her LDA implementation and Viktor Senderov for his ClaDS implementation.

Data-Availability Statement. The paper has an accompanying artifact that supports the evaluation: `https://zenodo.org/doi/10.5281/zenodo.10454311`.

References

1. The Miking benchmark suite. `https://github.com/miking-lang/miking-benchmarks` (2023), accessed: 2023-01-02
2. Miking DPPL. `https://github.com/miking-lang/miking-dppl` (2023), accessed: 2023-01-02
3. Appel, A.W.: Compiling with Continuations. Cambridge University Press (1991)
4. Asai, K., Uehara, C.: Selective cps transformation for shift and reset. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 40–52. Association for Computing Machinery (2017)
5. Baudart, G., Burroni, J., Hirzel, M., Mandel, L., Shinnar, A.: Compiling stan to generative probabilistic languages and extension to deep probabilistic programming. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 497–510. Association for Computing Machinery (2021)
6. Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N.D.: Pyro: Deep universal probabilistic programming. Journal of Machine Learning Research **20**(28), 1–6 (2019)
7. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003)
8. Borgström, J., Dal Lago, U., Gordon, A.D., Szymczak, M.: A lambda-calculus foundation for universal probabilistic programming. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 33–46. Association for Computing Machinery (2016)
9. Broman, D.: A vision of miking: Interactive programmatic modeling, sound language composition, and self-learning compilation. In: Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. pp. 55–60. Association for Computing Machinery (2019)
10. Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A.: Stan: A probabilistic programming language. Journal of Statistical Software, Articles **76**(1), 1–32 (2017)
11. Cusumano-Towner, M.F., Saad, F.A., Lew, A.K., Mansinghka, V.K.: Gen: A general-purpose probabilistic programming system with programmable inference. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 221–236. Association for Computing Machinery (2019)
12. Doucet, A., de Freitas, N., Gordon, N.: Sequential Monte Carlo Methods in Practice. Information Science and Statistics, Springer New York (2001)
13. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 237–247. Association for Computing Machinery (1993)
14. Funk, S., Kucharski, A.J., Camacho, A., Eggo, R.M., Yakob, L., Murray, L.M., Edmunds, W.J.: Comparative analysis of dengue and zika outbreaks reveals differences by setting and virus. PLOS Neglected Tropical Diseases **10**(12), 1–16 (2016)
15. Ge, H., Xu, K., Ghahramani, Z.: Turing: A language for flexible probabilistic inference. In: Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. vol. 84, pp. 1682–1690. Proceedings of Machine Learning Research (2018)

16. Gilks, W., Richardson, S., Spiegelhalter, D.: Markov Chain Monte Carlo in Practice. Chapman & Hall/CRC Interdisciplinary Statistics, Taylor & Francis (1995)
17. Goodman, N.D., Mansinghka, V.K., Roy, D., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence. pp. 220–229. AUAI Press (2008)
18. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages. `http://dippl.org` (2014), accessed: 2022-10-31
19. Goodman, N.D., Tenenbaum, J.B., Contributors, T.P.: Probabilistic Models of Cognition. `http://probmods.org/v2` (2016), accessed: 2022-06-10
20. Gothoskar, N., Cusumano-Towner, M., Zinberg, B., Ghavamizadeh, M., Pollok, F., Garrett, A., Tenenbaum, J., Gutfreund, D., Mansinghka, V.: 3DP3: 3D scene perception via probabilistic programming. In: Advances in Neural Information Processing Systems. vol. 34, pp. 9600–9612. Curran Associates, Inc. (2021)
21. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proceedings of the National academy of Sciences **101**(suppl_1), 5228–5235 (2004)
22. Hummelgren, L., Wikman, J., Eriksson, O., Haller, P., Broman, D.: Expression acceleration: Seamless parallelization of typed high-level languages. arXiv e-prints p. arXiv:2211.00621 (2022)
23. Jetz, W., Thomas, G.H., Joy, J.B., Hartmann, K., Mooers, A.O.: The global diversity of birds in space and time. Nature **491**(7424), 444–448 (2012)
24. Kudlicka, J., Murray, L.M., Ronquist, F., Schön, T.B.: Probabilistic programming for birth-death models of evolution using an alive particle filter with delayed sampling. In: Conference on Uncertainty in Artificial Intelligence (2019)
25. Kulkarni, T.D., Kohli, P., Tenenbaum, J.B., Mansinghka, V.: Picture: A probabilistic programming language for scene perception. In: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 4390–4399 (2015)
26. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 486–499. Association for Computing Machinery (2017)
27. Lew, A.K., Matheos, G., Zhi-Xuan, T., Ghavamizadeh, M., Gothoskar, N., Russell, S., Mansinghka, V.K.: Smcp3: Sequential monte carlo with probabilistic program proposals. In: Proceedings of The 26th International Conference on Artificial Intelligence and Statistics. pp. 7061–7088. Proceedings of Machine Learning Research (2023)
28. Lundén, D., Borgström, J., Broman, D.: Correctness of sequential monte carlo inference for probabilistic programming languages. In: Programming Languages and Systems. pp. 404–431. Springer International Publishing (2021)
29. Lundén, D., Çaylak, G., Ronquist, F., Broman, D.: Automatic alignment in higher-order probabilistic programming languages. In: Programming Languages and Systems. pp. 535–563 (2023)
30. Lundén, D., Öhman, J., Kudlicka, J., Senderov, V., Ronquist, F., Broman, D.: Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference. In: Programming Languages and Systems. pp. 29–56. Springer International Publishing (2022)
31. Lundén, D., Hummelgren, L., Kudlicka, J., Eriksson, O., Broman, D.: Suspension analysis and selective continuation-passing style for higher-order probabilistic programming languages. arXiv e-prints p. arXiv:2302.13051 (2024)
32. Maliet, O., Hartig, F., Morlon, H.: A model with many small shifts for estimating species-specific diversification rates. Nature Ecology & Evolution **3**(7), 1086–1092 (2019)

33. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3) (2012)
34. Murray, L., Lundén, D., Kudlicka, J., Broman, D., Schön, T.: Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In: Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics. vol. 84, pp. 1037–1046. Proceedings of Machine Learning Research (2018)
35. Murray, L.M.: Lazy object copy as a platform for population-based probabilistic programming. arXiv e-prints p. arXiv:2001.05293 (2020)
36. Murray, L.M., Schön, T.B.: Automated learning with a probabilistic programming language: Birch. Annual Reviews in Control **46**, 29–43 (2018)
37. Nee, S.: Birth-death models in macroevolution. Annual Review of Ecology, Evolution, and Systematics **37**(1), 1–17 (2006)
38. Nielsen, L.R.: A selective cps transformation. Electronic Notes in Theoretical Computer Science **45**, 311–331 (2001)
39. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
40. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: Proceedings of the 31st International Conference on Machine Learning. vol. 32, pp. 1935–1943. Proceedings of Machine Learning Research (2014)
41. Ritchie, D., Stuhlmüller, A., Goodman, N.: C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In: Proceedings of the 19th International Conference on Artificial Intelligence and Statistics. vol. 51, pp. 28–37. Proceedings of Machine Learning Research (2016)
42. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 317–328. Association for Computing Machinery (2009)
43. Ronquist, F., Kudlicka, J., Senderov, V., Borgström, J., Lartillot, N., Lundén, D., Murray, L., Schön, T.B., Broman, D.: Universal probabilistic programming offers a powerful approach to statistical phylogenetics. Communications Biology **4**(1), 244 (2021)
44. Ścibior, A., Kammar, O., Ghahramani, Z.: Functional programming for modular Bayesian inference. Proceedings of the ACM on Programming Languages **2**(ICFP) (2018)
45. Ścibior, A., Kammar, O., Vákár, M., Staton, S., Yang, H., Cai, Y., Ostermann, K., Moss, S.K., Heunen, C., Ghahramani, Z.: Denotational validation of higher-order Bayesian inference. Proceedings of the ACM on Programming Languages **2**(POPL) (2017)
46. Shivers, O.G.: Control-flow analysis of higher-order languages or taming lambda. Carnegie Mellon University (1991)
47. Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican. In: Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages. Association for Computing Machinery (2016)
48. Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.: Edward: A library for probabilistic modeling, inference, and criticism. arXiv e-prints p. arXiv:1610.09787 (2016)
49. Wingate, D., Stuhlmueller, A., Goodman, N.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. vol. 15, pp. 770–778. Proceedings of Machine Learning Research (2011)

50. Wood, F., Meent, J.W., Mansinghka, V.: A new approach to probabilistic programming inference. In: Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics. vol. 33, pp. 1024–1032. Proceedings of Machine Learning Research (2014)

Higher-Order LCTRSs and Their Termination

Liye Guo(✉) and Cynthia Kop

Radboud University, Nijmegen, The Netherlands
{l.guo,c.kop}@cs.ru.nl

Abstract Logically constrained term rewriting systems (LCTRSs) are a formalism for program analysis with support for data types that are not (co)inductively defined. Only imperative programs have been considered through the lens of LCTRSs so far since LCTRSs were introduced as a first-order formalism. In this paper, we propose logically constrained simply-typed term rewriting systems (LCSTRSs), a higher-order generalization of LCTRSs, which suits the needs of representing and analyzing functional programs. We also study the termination problem of LCSTRSs and define a variant of the higher-order recursive path ordering (HORPO) for the newly proposed formalism.

Keywords: Higher-order term rewriting · Constraints · Recursive path ordering.

1 Introduction

It is hardly a surprising idea that term rewriting can serve as a vehicle for reasoning about programs. During the last decade or so, the term rewriting community has seen a line of work that translates real-world problems from program analysis into questions about term rewriting systems, which include, for example, termination (see, e.g., [8,10,15,37]) and equivalence (see, e.g., [13,36,9]). Such applications take place across programming paradigms due to the versatile nature of term rewriting, and often materialize into automatable solutions.

Data types are a central building block of programs and must be properly handled in program analysis. While it is rarely a problem for term rewriting systems to represent (co)inductively defined data types, others such as integers and arrays traditionally require encoding; think of $\mathsf{neg}\ (\mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})))$ encoding -3. This usually turns out to cause more obfuscation than clarification to the methods applied and the results obtained. An alternative is to incorporate primitive data types into the formalism, which contributes to the proliferation of subtly different formalisms that are generally incompatible with each other, and it is often difficult to transfer techniques between such formalisms.

Logically constrained term rewriting systems (LCTRSs) [27,12] emerged from this proliferation as a unifying formalism seeking to be general in both the selection of primitive data types (little is presumed) and the applicability of varied methods (many are extensible). LCTRSs thus allow us to benefit from the broad term rewriting arsenal in a wide range of scenarios for program analysis

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 331–357, 2024.
https://doi.org/10.1007/978-3-031-57267-8_13

(see, e.g., [32,24,23]). In particular, rewriting induction on LCTRSs [12,30] offers a powerful tool for program verification.

As a first-order formalism, LCTRSs only naturally accommodate imperative programs. This paper aims to generalize this formalism in a higher-order setting.

Motivation. Below is a first-order LCTRS implementing the factorial function:

$$\mathsf{fact}\ n \to 1 \quad [n \leq 0] \qquad \mathsf{fact}\ n \to n * \mathsf{fact}\ (n-1) \quad [n > 0]$$

where $n \leq 0$ and $n > 0$ are logical constraints, which the integer n must satisfy respectively when the corresponding rewrite rule is applied. Suppose we have access to higher-order functions, a defining feature of functional programming; now we have the following alternative implementation of fact:

$$\mathsf{fact}\ n \to \mathsf{fold}\ (*)\ 1\ (\mathsf{genlist}\ n)$$
$$\mathsf{genlist}\ n \to \mathsf{nil} \quad [n \leq 0] \qquad \mathsf{genlist}\ n \to \mathsf{cons}\ n\ (\mathsf{genlist}\ (n-1)) \quad [n > 0]$$
$$\mathsf{fold}\ f\ y\ \mathsf{nil} \to y \qquad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)$$

Here fold takes an argument f, which itself represents a function. Higher-order functions such as fold do not fit into first-order LCTRSs, which leads to the first reason to generalize this formalism: to overcome the limitation of its expressivity.

There is another reason for higher-order LCTRSs. The latter implementation of fact reflects a pattern of functional programming: the combination of "standard" higher-order building blocks such as fold and map, and functions that are specific to the problem at hand. We note that a higher-order formalism can reveal more modularity in programs. It would be valuable to exploit such modularity in *analyses* as well.

With higher-order LCTRSs, we would like to explore automatable solutions to the termination problem of functional programs in the same fashion as the first-order case [27,25], or even better, to the finding of their complexity by term rewriting. Moreover, given two programs supposedly implementing the same function, a method that derives whether they are indeed equivalent is also desirable. For example, a proof that the above two implementations of fact are equivalent may serve as a correctness proof of the latter, less intuitive implementation (which in general might be an outcome of code refactoring). Such methods have been explored in a first-order setting [12,7].

Higher-order LCTRSs will broaden the horizons of both LCTRSs and higher-order term rewriting. The eventual goal is to have a formalism that can be deployed to analyze both imperative and functional programs, so that through this formalism, the abundant techniques based on term rewriting may be applied to automatic program analysis. This paper is a step toward that goal.

Contributions. The presentation begins with our perspective on higher-order term rewriting (without logical constraints) in Section 2. The contributions of this paper follow in subsequent sections:

- We propose the formalism of *logically constrained simply-typed term rewriting systems* (LCSTRSs), a higher-order generalization of LCTRSs, in Section 3.
- We adapt *reduction orderings* and *rule removal* to the newly proposed formalism, and define (as well as prove the soundness of) *constrained HORPO*—a variant of HORPO [21]—in Section 4. This includes changes to fit HORPO to curried notation and partial application, and to handle theory symbols and logical constraints in a similar way to RPO for first-order LCTRSs [27]. While this version of HORPO is not the most powerful higher-order termination technique, it offers a simple yet self-contained solution, and serves to illustrate how existing techniques may be extended.
- We have developed for our formalism the foundation of a new open-source analysis tool, in which an implementation of constrained HORPO is provided. It requires several new insights, especially with regard to the way theories and logical constraints are handled, and is discussed in Section 6.

2 Preliminaries

One of the first problems that a student of higher-order term rewriting faces is the absence of a standard formalism on which the literature agrees. This variety reflects the diverse interests and needs held by different authors.

In this section, we present simply-typed term rewriting systems (STRSs) [29] as the unconstrained basis of our formalism. This is one of the simplest higher-order formalisms, and closely resembles simple functional programs. We choose this formalism as our starting point because it is already powerful, while avoiding many of the complications that may be interesting for equational reasoning purposes but are not needed in program analysis, such as reduction modulo β.

Types and Terms. Types rule out undesired terms. We consider *simple types*: given a non-empty set $\mathcal{S}$ of *sorts* (or *base types*), the set $\mathcal{T}$ of simple types over $\mathcal{S}$ is generated by the grammar $\mathcal{T} ::= \mathcal{S} \mid (\mathcal{T} \to \mathcal{T})$. Right-associativity is assigned to $\to$ so we can omit some parentheses. The *order* of a type A, denoted by $\mathrm{ord}(A)$, is defined as follows: $\mathrm{ord}(A) = 0$ for $A \in \mathcal{S}$ and $\mathrm{ord}(A \to B) = \max(\mathrm{ord}(A) + 1, \mathrm{ord}(B))$.

Given disjoint sets $\mathcal{F}$ and $\mathcal{V}$, whose elements we call *function symbols* and *variables*, respectively, the set $\mathfrak{T}$ of *pre-terms* over $\mathcal{F}$ and $\mathcal{V}$ is generated by the grammar $\mathfrak{T} ::= \mathcal{F} \mid \mathcal{V} \mid (\mathfrak{T}\ \mathfrak{T})$. Left-associativity is assigned to the juxtaposition operation, called *application*, so $t_0\ t_1\ t_2$ stands for $((t_0\ t_1)\ t_2)$, for example.

We assume that every function symbol and variable is assigned a unique type. Typing works as expected: if pre-terms t_0 and t_1 have types $A \to B$ and A, respectively, $t_0\ t_1$ has type B. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$, denoted by $T(\mathcal{F}, \mathcal{V})$, is the subset of $\mathfrak{T}$ consisting of pre-terms with a type. We write $t : A$ if a term t has type A. The set of variables occurring in a term $t \in T(\mathcal{F}, \mathcal{V})$, denoted by $\mathrm{Var}(t)$, is defined as follows: $\mathrm{Var}(f) = \emptyset$ for $f \in \mathcal{F}$, $\mathrm{Var}(x) = \{\, x \,\}$ for $x \in \mathcal{V}$ and $\mathrm{Var}(t_0\ t_1) = \mathrm{Var}(t_0) \cup \mathrm{Var}(t_1)$. A term t is called *ground* if $\mathrm{Var}(t) = \emptyset$. The set of ground terms over $\mathcal{F}$ is denoted by $T(\mathcal{F}, \emptyset)$.

Substitutions and Contexts. Variables occurring in a term can be seen as placeholders: the occurrences of a variable may be replaced with terms which have the same type as the variable does. Type-preserving mappings from $\mathcal{V}$ to $T(\mathcal{F}, \mathcal{V})$ are called *substitutions.* Every substitution σ extends to a type-preserving mapping $\bar{\sigma}$ from $T(\mathcal{F}, \mathcal{V})$ to $T(\mathcal{F}, \mathcal{V})$. We write $t\sigma$ for $\bar{\sigma}(t)$ and define it as follows: $f\sigma = f$ for $f \in \mathcal{F}$, $x\sigma = \sigma(x)$ for $x \in \mathcal{V}$ and $(t_0\ t_1)\sigma = (t_0\sigma)\ (t_1\sigma)$.

Term formation gives rise to the concept of a context: a term containing a hole. Formally, let $\Box$ be a special terminal symbol denoting the hole, and the grammar $\mathfrak{C} ::= \Box \mid (\mathfrak{C}\ \mathfrak{T}) \mid (\mathfrak{T}\ \mathfrak{C})$ with the above rule for $\mathfrak{T}$ generates pre-terms containing exactly one occurrence of the hole. Given a type for the hole, a *context* is an element of $\mathfrak{C}$ which is typed as a term is. Let $C[]_A$ denote a context in which the hole has type A; filling the hole with a term $t : A$ produces the term $C[t]_A$ defined as follows: $\Box[t]_A = t$, $(C_0[]_A\ t_1)[t]_A = C_0[t]_A\ t_1$ and $(t_0\ C_1[]_A)[t]_A = t_0\ C_1[t]_A$. We usually omit types in the above notation, and in $C[t]$, t is understood as a term which has the same type as the hole does.

Rules and Rewriting. Now we have all the ingredients in our recipe for higher-order term rewriting. A *rewrite rule* $\ell \to r$ is an ordered pair of terms where ℓ and r have the same type, $\mathrm{Var}(\ell) \supseteq \mathrm{Var}(r)$ and ℓ assumes the form $f\ t_1 \cdots t_n$ for some function symbol f. Formally, a *simply-typed term rewriting system* (STRS) is a quadruple $(\mathcal{S}, \mathcal{F}, \mathcal{V}, \mathcal{R})$ where every element of $\mathcal{F} \cup \mathcal{V}$ is assigned a simple type over $\mathcal{S}$ and $\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ is a set of rewrite rules. We usually let $\mathcal{R}$ alone stand for the system and keep the details of term formation implicit.

The set $\mathcal{R}$ of rewrite rules induces the *rewrite relation* $\to_{\mathcal{R}} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$: $t \to_{\mathcal{R}} t'$ if and only if there exist a rewrite rule $\ell \to r \in \mathcal{R}$, a substitution σ and a context $C[]$ such that $t = C[\ell\sigma]$ and $t' = C[r\sigma]$. When there is no ambiguity about the system in question, we may simply write $\to$ for $\to_{\mathcal{R}}$.

Given a relation $\succ \subseteq X \times X$, an element x of X is called *terminating* with respect to $\succ$ if there is no infinite sequence $x = x_0 \succ x_1 \succ \cdots$, and $\succ$ is called *well-founded* if all the elements of X are terminating with respect to $\succ$. An STRS $\mathcal{R}$ is called terminating if $\to_{\mathcal{R}}$ is well-founded.

Example 1. The following rewrite rules constitute a terminating system:

$$\mathsf{take}\ \mathsf{zero}\ l \to \mathsf{nil} \quad \mathsf{take}\ n\ \mathsf{nil} \to \mathsf{nil} \quad \mathsf{take}\ (\mathsf{suc}\ n)\ (\mathsf{cons}\ x\ l) \to \mathsf{cons}\ x\ (\mathsf{take}\ n\ l)$$

where $\mathsf{zero} : \mathsf{nat}$, $\mathsf{suc} : \mathsf{nat} \to \mathsf{nat}$, $\mathsf{nil} : \mathsf{natlist}$, $\mathsf{cons} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ and $\mathsf{take} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ are function symbols, and $l : \mathsf{natlist}$, $n : \mathsf{nat}$ and $x : \mathsf{nat}$ are variables.

Example 2. The following rewrite rule constitutes a non-terminating system:

$$\mathsf{iterate}\ f\ x \to \mathsf{cons}\ x\ (\mathsf{iterate}\ f\ (f\ x))$$

where $\mathsf{cons} : \mathsf{nat} \to \mathsf{natlist} \to \mathsf{natlist}$ and $\mathsf{iterate} : (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{natlist}$ are function symbols, and $f : \mathsf{nat} \to \mathsf{nat}$ and $x : \mathsf{nat}$ are variables.

Limitations. The above formalism does not offer product types, polymorphism or λ-abstractions. What it does offer is its already expressive syntax enabling us, in a higher-order setting, to generalize LCTRSs and to discover what challenges one may face when extending existing unconstrained techniques. We expect that, once preliminary higher-order results are developed, we will adopt more features from other higher-order formalisms in future extensions.

The exclusion of λ-abstractions does not rid us of first-class functions, thanks to curried notation. For example, the occurrence of suc in $\mathsf{iterate}\ \mathsf{suc}\ \mathsf{zero}$ is partially (in this case, not at all) applied and still forms a term, which can be passed as an argument. Also, a term such as $\mathsf{iterate}\ (\lambda x.\, \mathsf{suc}\ (\mathsf{suc}\ x))\ \mathsf{zero}$ can be simulated at the cost of an extra rewrite rule (in this case, $\mathsf{add2}\ x \to \mathsf{suc}\ (\mathsf{suc}\ x)$). There are also straightforward ways of encoding product types.

Notions of Termination. If we combine the two systems from Examples 1 and 2, the outcome is surely non-terminating: $\mathsf{take}\ \mathsf{zero}\ (\mathsf{iterate}\ \mathsf{suc}\ \mathsf{zero})$ is not terminating, for example. From a Haskell programmer's perspective, however, this term is "terminating" due to the non-strictness of Haskell. In general, every functional language uses a certain evaluation strategy to choose a specific redex, if any, to rewrite within a term, whereas the rewrite relation we define in this section corresponds to full rewriting: the redex is chosen non-deterministically.

Furthermore, programmers usually care only about the termination of terms that are reachable from the entry point of a program and seldom consider full termination: the termination of all terms, i.e., the well-foundedness of the rewrite relation. We study full termination with respect to full rewriting in this paper, as it implies any other termination properties and full termination is often a prerequisite for determining properties such as confluence and equivalence.

3 Logically Constrained STRSs

Term rewriting systems do not have primitive data types built in; with some function symbols *constructing* (introducing) values of a certain type and pattern matching rules *deconstructing* (eliminating) those values, a term rewriting system relies on (co)inductively defined data types. While (co)inductive reasoning is straightforward this way, data types such as integers and arrays require encoding, which can be convoluted; think of the space-consuming unary representation of a number or a binary representation which takes less space but shifts the burden to rewrite rules defining arithmetic, and negative numbers bring up even more complications. Besides, such encoding neglects advances in modern SMT solvers.

In this section, we extend unconstrained STRSs with logical constraints so that data types that are not (co)inductively defined can be represented directly, and analysis tools can take advantage of existing SMT solvers. We follow the ideas of first-order LCTRSs [27,12]. Specifically, we will consider systems over *arbitrary* first-order theories, i.e., we are not bound to, say, systems over integers, while avoiding higher-order logical constraints. In the unconstrained part of such a system (outside theories), however, higher-order arguments and results are still completely usable.

3.1 Terms Modulo Theories

Following Section 2, we postulate a set $\mathcal{S}$ of sorts, a set $\mathcal{F}$ of function symbols and a set $\mathcal{V}$ of variables where every element of $\mathcal{F} \cup \mathcal{V}$ is assigned a simple type over $\mathcal{S}$. First, we assume that there is a distinguished subset $\mathcal{S}_\vartheta$ of $\mathcal{S}$, called the set of *theory sorts*. The grammar $\mathcal{T}_\vartheta ::= \mathcal{S}_\vartheta \mid (\mathcal{S}_\vartheta \to \mathcal{T}_\vartheta)$ generates the set $\mathcal{T}_\vartheta$ of *theory types* over $\mathcal{S}_\vartheta$. Note that the order of a theory type is never greater than one. Next, we assume that there is a distinguished subset $\mathcal{F}_\vartheta$ of $\mathcal{F}$, called the set of *theory symbols*, and that the type of every theory symbol is in $\mathcal{T}_\vartheta$, which means that the type of any argument passed to a theory symbol is a theory sort. Elements of $T(\mathcal{F}_\vartheta, \mathcal{V})$ are called *theory terms*. Last, for technical reasons, we assume that there are infinitely many variables of each type.

Theory symbols are interpreted in an underlying theory: given an $\mathcal{S}_\vartheta$-indexed family of sets $(\mathfrak{X}_A)_{A \in \mathcal{S}_\vartheta}$, we extend it to a $\mathcal{T}_\vartheta$-indexed family by letting $\mathfrak{X}_{A \to B}$ be the set of mappings from $\mathfrak{X}_A$ to $\mathfrak{X}_B$; an *interpretation* of theory symbols is a $\mathcal{T}_\vartheta$-indexed family of mappings $(\llbracket\cdot\rrbracket_A)_{A \in \mathcal{T}_\vartheta}$ where $\llbracket\cdot\rrbracket_A$ assigns to each theory symbol of type A an element of $\mathfrak{X}_A$ and is bijective[1] if $A \in \mathcal{S}_\vartheta$. Theory symbols whose type is a theory sort are called *values*. Given an interpretation of theory symbols $(\llbracket\cdot\rrbracket_A)_{A \in \mathcal{T}_\vartheta}$, we extend each indexed mapping $\llbracket\cdot\rrbracket_B$ to one that assigns to each *ground theory term* of type B an element of $\mathfrak{X}_B$ by letting $\llbracket t_0\ t_1\rrbracket_B$ be $\llbracket t_0\rrbracket_{A \to B}(\llbracket t_1\rrbracket_A)$. We usually write just $\llbracket\cdot\rrbracket$ when the type can be deduced.

Example 3. Let $\mathcal{S}_\vartheta$ be $\{\,\mathsf{int}\,\}$. Then $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ is a theory type over $\mathcal{S}_\vartheta$ while $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ is not. Let $\mathcal{F}_\vartheta$ be $\{\,\mathsf{sub}\,\} \cup \{\,\bar{n} \mid n \in \mathbb{Z}\,\}$ where $\mathsf{sub} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $\bar{n} : \mathsf{int}$. The values are the elements of $\{\,\bar{n} \mid n \in \mathbb{Z}\,\}$. Let $\mathfrak{X}_{\mathsf{int}}$ be $\mathbb{Z}$, $\llbracket\cdot\rrbracket_{\mathsf{int}}$ be the mapping $\bar{n} \mapsto n$ and $\llbracket\mathsf{sub}\rrbracket$ be the mapping $\lambda m.\, \lambda n.\, m - n$. The interpretation of $\mathsf{sub}\ \bar{1}$ is the mapping $\lambda n.\, 1 - n$.

We are not limited to the theory of integers:

Example 4. To reason about integer arrays, we could either represent them as lists and simulate random access through more costly list traversal (which affects the complexity), or consider a theory of bounded arrays as follows: Let $\mathcal{S}_\vartheta$ be $\{\,\mathsf{int}, \mathsf{intarray}\,\}$ and $\mathcal{F}_\vartheta$ be the union of $\{\,\mathsf{size}, \mathsf{select}, \mathsf{store}\,\}$, $\{\,\bar{n} \mid n \in \mathbb{Z}\,\}$ and $\{\,\langle \bar{n}_0, \ldots, \bar{n}_{k-1}\rangle \mid k \in \mathbb{N} \text{ and } \forall i.\, n_i \in \mathbb{Z}\,\}$ where $\mathsf{size} : \mathsf{intarray} \to \mathsf{int}$, $\mathsf{select} : \mathsf{intarray} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{store} : \mathsf{intarray} \to \mathsf{int} \to \mathsf{int} \to \mathsf{intarray}$, $\bar{n} : \mathsf{int}$ and $\langle \bar{n}_0, \ldots, \bar{n}_{k-1}\rangle : \mathsf{intarray}$. Let $\mathfrak{X}_{\mathsf{int}}$ and $\mathfrak{X}_{\mathsf{intarray}}$ be $\mathbb{Z}$ and $\mathbb{Z}^*$, respectively. Let $\llbracket\cdot\rrbracket_{\mathsf{int}}$ be the mapping $\bar{n} \mapsto n$ and $\llbracket\cdot\rrbracket_{\mathsf{intarray}}$ be the mapping $\langle \bar{n}_0, \ldots, \bar{n}_{k-1}\rangle \mapsto n_0 \ldots n_{k-1}$. Let $\llbracket\mathsf{size}\rrbracket(n_0 \ldots n_{k-1})$ be k. Let $\llbracket\mathsf{select}\rrbracket(n_0 \ldots n_{k-1}, i)$ be n_i if $0 \le i < k$, and 0 otherwise. Let $\llbracket\mathsf{store}\rrbracket(n_0 \ldots n_{k-1}, i, m)$ be $n_0 \ldots n_{i-1} m n_{i+1} \ldots n_{k-1}$ if $0 \le i < k$, and $n_0 \ldots n_{k-1}$ otherwise. Note that the values include *theory symbols* $\langle \bar{n}_0, \ldots, \bar{n}_{k-1}\rangle : \mathsf{intarray}$ as well as $\bar{n} : \mathsf{int}$.

[1] The bijectivity is assumed so that values (see below) are isomorphic to (and therefore a representation of) elements of $(\mathfrak{X}_A)_{A \in \mathcal{S}_\vartheta}$.

In this paper, we largely consider the theory of integers in Example 3 when giving examples because it is easy to understand. This particular theory does not play a special role for the formalism we will shortly present; in fact, the theory of *bit vectors* may be more appropriate to real-world programs using integers, and our formalism is not biased toward any choice of theories. In particular, we do not have to choose predefined theories from SMT-LIB [3]. The theory of bounded arrays in Example 4 is an instance of such a "non-standard" theory (which can nevertheless be encoded within the theory of functional arrays). On the other hand, theories supported by SMT solvers are preferable in light of automation.

3.2 Constrained Rewriting

Constrained rewriting requires the theory sort bool: we henceforth assume that $\mathsf{bool} \in \mathcal{S}_\vartheta$, $\{\mathfrak{f}, \mathfrak{t}\} \subseteq \mathcal{F}_\vartheta$, $\mathfrak{X}_{\mathsf{bool}} = \{0, 1\}$, $[\![\mathfrak{f}]\!]_{\mathsf{bool}} = 0$ and $[\![\mathfrak{t}]\!]_{\mathsf{bool}} = 1$. A *logical constraint* is a theory term φ such that φ has type bool and the type of each variable in $\mathrm{Var}(\varphi)$ is a theory sort. A (constrained) *rewrite rule* is a triple $\ell \to r\ [\varphi]$ where ℓ and r are terms which have the same type, φ is a logical constraint, the type of each variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$ is a theory sort and ℓ is a term that assumes the form $f\ t_1 \cdots t_n$ for some function symbol f and contains at least one function symbol in $\mathcal{F} \setminus \mathcal{F}_\vartheta$.[2]

This definition can be obscure at first glance, especially when compared with its unconstrained counterpart in Section 2: variables which do not occur in ℓ are allowed to occur in r, not to mention the logical constraint φ as a brand-new component. Given a rewrite rule $\ell \to r\ [\varphi]$, the idea is that variables occurring in φ are to be instantiated to values which make φ true and other variables which occur in r but not in ℓ are to be instantiated to arbitrary values—note that the type of each of these variables is a theory sort. Formally, given an interpretation of theory symbols $[\![\cdot]\!]$, a substitution σ is said to *respect* a rewrite rule $\ell \to r\ [\varphi]$ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi) \cup (\mathrm{Var}(r) \setminus \mathrm{Var}(\ell))$ and $[\![\varphi\sigma]\!] = 1$.

We summarize all the above ingredients in the following definition:

Definition 1. *A* logically constrained STRS *(LCSTRS) consists of $\mathcal{S}$, $\mathcal{S}_\vartheta$, $\mathcal{F}$, $\mathcal{F}_\vartheta$, $\mathcal{V}$, $(\mathfrak{X}_A)$, $[\![\cdot]\!]$ and $\mathcal{R}$ where*

1. *$\mathcal{S}$ is a set of sorts,*
2. *$\mathcal{S}_\vartheta \subseteq \mathcal{S}$ is a set of theory sorts which contains* bool,
3. *$\mathcal{F}$ is a set of function symbols in which every function symbol is assigned a simple type over $\mathcal{S}$,*
4. *$\mathcal{F}_\vartheta \subseteq \mathcal{F}$ is a set of theory symbols in which the type of every theory symbol is a theory type over $\mathcal{S}_\vartheta$, with* $\mathfrak{f} : \mathsf{bool}$ *and* $\mathfrak{t} : \mathsf{bool}$ *elements of $\mathcal{F}_\vartheta$,*
5. *$\mathcal{V}$ is a set of variables disjoint from $\mathcal{F}$ in which every variable is assigned a simple type over $\mathcal{S}$ and there are infinitely many variables to which every type is assigned,*

[2] We do not require f to be in $\mathcal{F} \setminus \mathcal{F}_\vartheta$ (that is, f can be a theory symbol) because a theory symbol may occur at the head position of a rewrite rule's left-hand side in rewriting induction, and this general definition is in line with first-order LCTRSs.

6. *$(\mathfrak{X}_A)$ is an $\mathcal{S}_\vartheta$-indexed family of sets such that $\mathfrak{X}_{\mathsf{bool}} = \{0, 1\}$,*
7. *$[\![\cdot]\!]$ is an interpretation of theory symbols such that $[\![\mathfrak{f}]\!] = 0$ and $[\![\mathfrak{t}]\!] = 1$, and*
8. *$\mathcal{R} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}_\vartheta, \mathcal{V})$ is a set of rewrite rules.*

We usually let $\mathcal{R}$ alone stand for the system.

And the following definition concludes the elaboration of constrained rewriting:

Definition 2. *Given an LCSTRS $\mathcal{R}$, the set of rewrite rules induces the* rewrite relation *$\rightarrow_{\mathcal{R}} \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ such that $t \rightarrow_{\mathcal{R}} t'$ if and only if one of the following conditions is true:*

1. *There exist a rewrite rule $\ell \rightarrow r\ [\varphi] \in \mathcal{R}$, a substitution σ which respects $\ell \rightarrow r\ [\varphi]$ and a context $C[]$ such that $t = C[\ell\sigma]$ and $t' = C[r\sigma]$.*
2. *There exist theory symbols $v_1 : A_1, \ldots, v_n : A_n, v' : B$ and $f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$ for $n > 0$ and $A_1, \ldots, A_n, B \in \mathcal{S}_\vartheta$ such that $[\![f\ v_1 \cdots v_n]\!] = [\![v']\!]$, and a context $C[]$ such that $t = C[f\ v_1 \cdots v_n]$ and $t' = C[v']$.*

Note that the above conditions are mutually exclusive for any given context $C[]$: $f\ v_1 \cdots v_n$ is a theory term, whereas ℓ in any rewrite rule $\ell \rightarrow r\ [\varphi]$ is not. If $t \rightarrow_{\mathcal{R}} t'$ due to the second condition, we also write $t \rightarrow_\kappa t'$ and call it a step of calculation. *When no ambiguity arises, we may simply write $\rightarrow$ for $\rightarrow_{\mathcal{R}}$.*

Example 5. We can rework Example 1 into an LCSTRS:

$$\begin{array}{rll} \mathsf{take}\ n\ l \rightarrow \mathsf{nil} & [n \leq 0] & \mathsf{take}\ n\ \mathsf{nil} \rightarrow \mathsf{nil} \\ \mathsf{take}\ n\ (\mathsf{cons}\ x\ l) \rightarrow \mathsf{cons}\ x\ (\mathsf{take}\ (n-1)\ l) & [n > 0] & \end{array}$$

where $\mathcal{S} = \mathcal{S}_\vartheta \cup \{\mathsf{intlist}\}$, $\mathcal{S}_\vartheta = \{\mathsf{bool}, \mathsf{int}\}$, $\mathcal{F} = \mathcal{F}_\vartheta \cup \{\mathsf{nil}, \mathsf{cons}, \mathsf{take}\}$, $\mathcal{F}_\vartheta = \{\leq, >, -, \mathfrak{f}, \mathfrak{t}\} \cup \mathbb{Z}$, $\mathcal{V} \supseteq \{l, n, x\}$, $\leq : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{bool}$, $> : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{bool}$, $- : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$, $\mathfrak{f} : \mathsf{bool}$, $\mathfrak{t} : \mathsf{bool}$, $v : \mathsf{int}$ for all $v \in \mathbb{Z}$, $\mathsf{nil} : \mathsf{intlist}$, $\mathsf{cons} : \mathsf{int} \rightarrow \mathsf{intlist} \rightarrow \mathsf{intlist}$, $\mathsf{take} : \mathsf{int} \rightarrow \mathsf{intlist} \rightarrow \mathsf{intlist}$, $l : \mathsf{intlist}$, $n : \mathsf{int}$ and $x : \mathsf{int}$.

Here and henceforth we let integer literals and operators, e.g., 0, 1, $\leq$, $>$ and $-$, denote both the corresponding theory symbols and their respective images under the interpretation—in contrast to Examples 3 and 4, where we pedantically make a distinction between, say, $\bar{1}$ and 1. We also use infix notation for some binary operators to improve readability, and omit the logical constraint of a rewrite rule when it is $\mathfrak{t}$. Below is a rewrite sequence:

$$\begin{array}{l} \mathsf{take}\ 1\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ l)) \rightarrow \mathsf{cons}\ x\ (\mathsf{take}\ (1-1)\ (\mathsf{cons}\ y\ l)) \\ \rightarrow_\kappa \mathsf{cons}\ x\ (\mathsf{take}\ 0\ (\mathsf{cons}\ y\ l)) \rightarrow \mathsf{cons}\ x\ \mathsf{nil} \end{array}$$

Example 6. In Section 1, the rewrite rules implementing the factorial function by fold constitute an LCSTRS. Below is a rewrite sequence:

$$\begin{array}{l} \mathsf{fact}\ 1 \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{genlist}\ 1) \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ (\mathsf{genlist}\ (1-1))) \\ \quad \rightarrow_\kappa \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ (\mathsf{genlist}\ 0)) \rightarrow \mathsf{fold}\ (*)\ 1\ (\mathsf{cons}\ 1\ \mathsf{nil}) \\ \quad \rightarrow (*)\ 1\ (\mathsf{fold}\ (*)\ 1\ \mathsf{nil}) \rightarrow (*)\ 1\ 1 \rightarrow_\kappa 1 \end{array}$$

Example 7. Consider the rewrite rule $\mathsf{readint} \to n$, in which the variable $n : \mathsf{int}$ occurs on the right-hand side of $\to$ but not on the left. Unconstrained STRSs do not permit such a rewrite rule, but LCSTRSs do. It looks as if we might rewrite $\mathsf{readint}$ to a variable but it is not the case: all the substitutions which respect this rewrite rule must map n to a value. Indeed, $\mathsf{readint}$ is always rewritten to a value of type int. We may have, say, $\mathsf{readint} \to 42$. Such variables can be used to model user input.

Example 8. Getting input by means of the rewrite rule from Example 7 has one flaw: in case of multiple integers to be read, the order of reading each is non-deterministic. Even in the presence of an evaluation strategy, the order may not be the desired one. We can use continuation-passing style to choose an order:

$$\mathsf{readint}\ k \to k\ n \quad \mathsf{comp}\ g\ f\ x \to g\ (f\ x) \quad \mathsf{sub} \to \mathsf{readint}\ (\mathsf{comp}\ \mathsf{readint}\ (-))$$

where $\mathsf{comp} : ((\mathsf{int} \to \mathsf{int}) \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$. If the first and the second integers to be read were 1 and 2, respectively, the following rewrite sequence would be the only one starting from sub:

$$\begin{aligned}\mathsf{sub} &\to \mathsf{readint}\ (\mathsf{comp}\ \mathsf{readint}\ (-)) \to \mathsf{comp}\ \mathsf{readint}\ (-)\ 1\\ &\to \mathsf{readint}\ ((-)\ 1) \to (-)\ 1\ 2 \to_\kappa -1\end{aligned}$$

Since there is no way to specify the actual input within an LCSTRS, rewrite sequences such as the one above cannot be derived deterministically. Nevertheless, this example demonstrates that the newly proposed formalism can represent relatively sophisticated control mechanisms utilized by functional programs.

Remarks. We reflect on some of the concepts presented in this section:

– We use the phrase "terms modulo theories" in line with "satisfiability modulo theories": some function symbols are interpreted within a theory. While such an interpretation gives rise to a way of identifying certain terms, namely those that are convertible to each other with respect to $\to_\kappa$, we do not consider them identified (in other words, modulo κ) in this paper.
– First-order LCTRSs can be seen as instances of the newly proposed formalism, i.e., ones in which the type order of each function symbol is no greater than one, variables with a non-zero type order (i.e., higher-order variables) are excluded, and the type of both sides of a rewrite rule is always a sort.
– Logical constraints are essentially first-order: the type order of a theory symbol cannot be greater than one while higher-order variables are excluded. This restriction rules out, for example, the following implementation:

$$\begin{array}{lll}\mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{cons}\ x\ (\mathsf{filter}\ f\ l) & [f\ x] & \mathsf{filter}\ f\ \mathsf{nil} \to \mathsf{nil}\\ \mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{filter}\ f\ l & [\neg\ (f\ x)] & \end{array}$$

The filter function can actually be implemented in an LCSTRS as follows:

$$\begin{gathered}\mathsf{filter}\ f\ (\mathsf{cons}\ x\ l) \to \mathsf{if}\ (f\ x)\ (\mathsf{cons}\ x\ (\mathsf{filter}\ f\ l))\ (\mathsf{filter}\ f\ l)\\ \mathsf{filter}\ f\ \mathsf{nil} \to \mathsf{nil} \qquad \mathsf{if}\ \mathfrak{t}\ l\ l' \to l \qquad \mathsf{if}\ \mathfrak{f}\ l\ l' \to l'\end{gathered}$$

In the former implementation, the problem is not the higher-order variable f itself but its occurrence in logical constraints. In this case, because the filter function is usually meant to be used in combination with "user-defined" predicates—which are function symbols defined by rewrite rules and therefore do not belong to the theories—it makes sense to disallow f from occurring in logical constraints. In general, we may encounter use cases for higher-order constraints; until then, we focus on first-order constraints, which are very common in functional programs.

4 Constrained Higher-Order Recursive Path Ordering

Recall that an important part of our goal is to allow the abundant term rewriting techniques to be applied toward program analysis. We have defined a formalism for constrained higher-order term rewriting; now it remains to be seen that—or *how*—existing techniques can be extended to it.

In the rest of this paper, we consider termination, an important aspect of program analysis and a topic that has been studied by the term rewriting community for decades. Not only is termination itself critical to the correctness of certain programs, but it also facilitates other analyses by admitting well-founded induction on terms.

In this section, we adapt HORPO [21] to our formalism. This is one of the oldest, yet still effective techniques for higher-order termination. HORPO can be used either as a stand-alone method or in a higher-order version of the dependency pair framework [1,39,11,25]. Hence, this adaptation offers a solid basis for use in an analysis tool's termination module. We will discuss the use of HORPO within the dependency pair framework in Section 5, and its automation in Section 6.

Constrained RPO for first-order LCTRSs was proposed in [27]. We take inspiration from it for its approach to theory terms, formalize the ideas, and add support for (higher) types as well as partial application.

4.1 HORPO, Unconstrained and Uncurried

We first recall HORPO in its original form. Note that the original definition is based on an unconstrained and uncurried format, and a thorough discussion on it is beyond the scope of this paper. The following presentation is mostly informal and only serves the purposes of comparison and inspiration.

We begin with two standard definitions:

Definition 3. *Given relations* $\succsim$ *and* $\succ$ *over* X*, the* generalized lexicographic ordering $\succ^{\mathfrak{l}} \subseteq X^* \times X^*$ *is induced as follows:* $x_1 \dots x_m \succ^{\mathfrak{l}} y_1 \dots y_n$ *if and only if there exists* $k \leq \min(m, n)$ *such that* $x_i \succsim y_i$ *for all* $i < k$ *and* $x_k \succ y_k$*.*

Definition 4. *Given relations* $\succsim$ *and* $\succ$ *over* X*, the* generalized multiset ordering $\succ^{\mathfrak{m}} \subseteq X^* \times X^*$ *is induced as follows:* $x_1 \dots x_m \succ^{\mathfrak{m}} y_1 \dots y_n$ *if and only if there exist a non-empty subset* I *of* $\{1, \dots, m\}$ *and a mapping* π *from* $\{1, \dots, n\}$ *to* $\{1, \dots, m\}$ *such that*

1. $\forall i \in I. \forall j \in \pi^{-1}(i). x_i \succ y_j$,
2. $\forall i \in \{1, \ldots, m\} \setminus I. \forall j \in \pi^{-1}(i). x_i \gtrsim y_j$, *and*
3. $\forall i \in \{1, \ldots, m\} \setminus I. \left|\pi^{-1}(i)\right| = 1$.

Here the generalized multiset ordering is formulated in terms of lists because we will compare argument lists by this ordering and this formulation facilitates implementation. In the following definition of HORPO, when we refer to the above definitions, $\gtrsim$ is the *equality* over terms and $\succ$ is HORPO itself.

Roughly, HORPO extends a given ordering over function symbols, and when considering terms headed by the same function symbol, compares the arguments by either of the above orderings. Given a well-founded ordering $\blacktriangleright \subseteq \mathcal{F} \times \mathcal{F}$, called the *precedence*, and a mapping $\mathfrak{s} : \mathcal{F} \to \{\mathfrak{l}, \mathfrak{m}\}$, called the *status*, HORPO is a type-preserving relation $\succ$ such that $s \succ t$ if and only if one of the following conditions is true:

1. $s = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$ and $\exists k. s_k \succeq t$.
2. $s = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$, $t = @(\ldots @(@(t_0, t_1), t_2) \ldots, t_n)$ and $\forall i. s \succ t_i \lor \exists k. s_k \succeq t_i$.
3. $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \in \mathcal{F}$, $g \in \mathcal{F}$, $f \blacktriangleright g$ and $\forall i. s \succ t_i \lor \exists k. s_k \succeq t_i$.
4. $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \ldots s_m \succ^{\mathfrak{l}} t_1 \ldots t_m$ and $\forall i. s \succ t_i \lor \exists k. s_k \succeq t_i$.
5. $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $f \in \mathcal{F}$, $\mathfrak{s}(f) = \mathfrak{m}$ and $s_1 \ldots s_m \succ^{\mathfrak{m}} t_1 \ldots t_m$.
6. $s = @(s_0, s_1)$, $t = @(t_0, t_1)$ and $s_0 s_1 \succ^{\mathfrak{m}} t_0 t_1$.
7. $s = \lambda x. s_0$, $t = \lambda x. t_0$ and $s_0 \succ t_0$.

Here $\succeq$ denotes the reflexive closure of $\succ$.

We call this format uncurried because every function symbol has an arity, i.e., the number of arguments guaranteed for each occurrence of the function symbol in a term. This is indicated by the functional notation $f(s_1, \ldots, s_m)$ as opposed to $f\ s_1 \cdots s_m$. If f has arity m, its occurrence in a term must take m arguments so $f(s_1, \ldots, s_{m-1})$ is not a well-formed term, for example. A function symbol's type (or more technically, its type declaration) can permit more arguments than its arity guarantees. Such an extra argument is supplied through the syntactic form $@(\cdot, \cdot)$. For example, if the same function symbol f is given an extra argument s_{m+1}, we write $@(f(s_1, \ldots, s_m), s_{m+1})$. This syntactic form is also used to pass arguments to variables and λ-abstractions.

The difference between an uncurried and a curried format is more than a notational issue, and poses technical challenges to our extension of HORPO. Another source of challenges is, as one would expect, constrained rewriting.

4.2 Rule Removal

HORPO is defined as a reduction ordering $\succ$, which is a type-preserving, *stable* (i.e., $t \succ t'$ implies $t\sigma \succ t'\sigma$), *monotonic* (i.e., $t \succ t'$ implies $C[t] \succ C[t']$) and well-founded relation. Note that despite its name, HORPO is not necessarily transitive. If such a relation *orients* all the rewrite rules in $\mathcal{R}$ (i.e., $\ell \succ r$ for all $\ell \to r \in \mathcal{R}$), we can conclude that the rewrite relation $\to_{\mathcal{R}}$ is well-founded.

A similar strategy for LCSTRSs requires a few tweaks. First, stability should be tightly coupled with rule orientation because every rewrite rule now is equipped with a logical constraint, which decides what substitutions are expected when the rewrite rule is applied. Second, the monotonicity requirement can be weakened because ℓ is never a theory term in a rewrite rule $\ell \to r\ [\varphi]$. We define as follows:

Definition 5. *A type-preserving relation $\Rightarrow \subseteq T(\mathcal{F}, \mathcal{V}) \times T(\mathcal{F}, \mathcal{V})$ is said*

1. *to* stably orient *a rewrite rule $\ell \to r\ [\varphi]$ if $\ell\sigma \Rightarrow r\sigma$ for each substitution σ which respects the rewrite rule, and*
2. *to be* rule-monotonic *if $t \Rightarrow t'$ implies $C[t] \Rightarrow C[t']$ when $t \notin T(\mathcal{F}_\vartheta, \mathcal{V})$.*

Besides having rewrite rules stably oriented, we need to deal with calculation. It turns out to be unnecessary to search for a well-founded relation which includes $\to_\kappa$, given the following observation:

Lemma 1. *$\to_\kappa$ is well-founded.*

Proof. The term size strictly decreases through every step of calculation. □

We rather look for a type-preserving and well-founded relation $\succ$ which stably orients every rewrite rule, is rule-monotonic, and is *compatible* with $\to_\kappa$, i.e., $\to_\kappa ; \succ \subseteq \succ^+$ or $\succ ; \to_\kappa \subseteq \succ^+$. This strategy is an instance of *rule removal*:

Theorem 1. *Given an LCSTRS $\mathcal{R}$, the rewrite relation $\to_{\mathcal{R}}$ is well-founded if and only if there exist sets $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\to_{\mathcal{R}_1}$ is well-founded and $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$, and type-preserving, rule-monotonic relations $\Rightarrow$ and $\succ$ such that*

1. *$\Rightarrow$ includes $\to_\kappa$ and stably orients every rewrite rule in $\mathcal{R}_1$,*
2. *$\succ$ is well-founded and stably orients every rewrite rule in $\mathcal{R}_2$, and*
3. *$\Rightarrow ; \succ \subseteq \succ^+$ or $\succ ; \Rightarrow \subseteq \succ^+$.*

Here $\to_{\mathcal{R}_1}$ assumes the same term formation and interpretation as $\to_{\mathcal{R}}$ does.

Proof. If $\to_{\mathcal{R}}$ is well-founded, take $\mathcal{R}_1 = \emptyset$, $\mathcal{R}_2 = \mathcal{R}$, $\Rightarrow\, = \,\to_\kappa$ and $\succ\, = \,\to_{\mathcal{R}}$. Note that $\to_\emptyset = \to_\kappa$ by definition.

Now assume given $\mathcal{R}_1$, $\mathcal{R}_2$, $\Rightarrow$ and $\succ$. Since $\Rightarrow$ is rule-monotonic, includes $\to_\kappa$ and stably orients every rewrite rule in $\mathcal{R}_1$, $\to_{\mathcal{R}_1} \subseteq \Rightarrow$. So the compatibility of $\succ$ with $\Rightarrow$ implies its compatibility with $\to_{\mathcal{R}_1}$, which in turn implies the well-foundedness of $\to_{\mathcal{R}_1} \cup \succ$, given that both $\to_{\mathcal{R}_1}$ and $\succ$ are well-founded. Since $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$ and $\succ$ is a rule-monotonic relation which stably orients every rewrite rule in $\mathcal{R}_2$, $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}_1} \cup \succ$. Hence, $\to_{\mathcal{R}}$ is well-founded. □

In a termination proof of $\mathcal{R}$, Theorem 1 allows us to remove rewrite rules that are in $\mathcal{R}_2$ from $\mathcal{R}$. If none of the rewrite rules are left after iterations of rule removal, the termination of the original system can be concluded with Lemma 1.

4.3 Constrained HORPO for LCSTRSs

Before adapting HORPO for LCSTRSs, we discuss how the theories may be handled. Let us consider the following system:

$$\mathsf{rec}\ n\ x\ f \to x \quad [n \leq 0] \qquad \mathsf{rec}\ n\ x\ f \to f\ (n-1)\ (\mathsf{rec}\ (n-1)\ x\ f) \quad [n > 0]$$

where $\mathsf{rec} : \mathsf{int} \to \mathsf{int} \to (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$. In the second rewrite rule, the left-hand side of $\to$ is $\mathsf{rec}\ n\ x\ f$ while the right-hand side has a subterm $\mathsf{rec}\ (n-1)\ x\ f$. It is natural to expect $n \succ n - 1$ in the construction of HORPO. Note that this is impossible with respect to any recursive path ordering for unconstrained rewriting because n is a variable occurring in $n - 1$; in an unconstrained setting, we actually have $n - 1 \succ n$. Hence, we must somehow take the logical constraint $n > 0$ into account. To this end, we largely follow the ideas of constrained RPO for first-order LCTRSs [27].

The occurrence of n in the logical constraint ensures that n is instantiated to a value, say 42, when the rewrite rule is applied, and it is sensible to have $42 \succ 42 - 1$. Also, $n > 0$ guarantees that all the sequences of such descents are finite, i.e., the ordering $\lambda m.\, \lambda n.\, m > 0 \wedge m > n$, denoted by $\sqsupset$, is well-founded. Let $\varphi \models \varphi'$ denote, on the assumption that φ and φ' are logical constraints such that $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$, that $[\![\varphi\sigma]\!] = 1$ implies $[\![\varphi'\sigma]\!] = 1$ for each substitution σ which maps variables in $\mathrm{Var}(\varphi)$ to values. Then we have $n > 0 \models n \sqsupset n - 1$. We thus would like to have $s \succ t$ if $\varphi \models s \sqsupset t$.

However, with the same ordering $\sqsupset$, we have both $m > 0 \wedge m > n \models m \sqsupset n$ and $n > 0 \wedge n > m \models n \sqsupset m$, whereas we cannot have both $m \succ n$ and $n \succ m$ without breaking the well-foundedness of $\succ$. To resolve this issue, we split $\succ$ into a family of relations $(\succ_\varphi)$ indexed by logical constraints, and let $s \succ_\varphi t$ be true if $\varphi \models s \sqsupset t$. We also introduce a separate family of relations $(\succsim_\varphi)$ such that $s \succsim_\varphi t$ if $\varphi \models s \sqsupseteq t$ where $\sqsupseteq$ is the reflexive closure of $\sqsupset$. Hence, $\succsim_\varphi$ is *not* necessarily the reflexive closure of $\succ_\varphi$; if it was, even $n \succsim_{n \geq 1} 1$ would not be obtainable.

Now we have a family of pairs $(\succsim_\varphi, \succ_\varphi)$, which does not seem to suit rule removal; after all, the essential requirement is a fixed relation which is type-preserving, rule-monotonic, well-founded and at least compatible with $\to_\kappa$. When the definition of constrained HORPO is fully presented, we will show that $\succ_\mathfrak{t}$—the irreflexive relation indexed by the boolean $\mathfrak{t}$—is such a relation and stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succ_\varphi r$.

The annotation φ of HORPO does not capture variables in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$, which also have a part to play in the decision of what substitutions are expected when $\ell \to r\ [\varphi]$ is applied. We may use a new annotation to accommodate these variables but there is a hack (also present in [38]): given a variable in $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell)$, it can be harmlessly appended to φ, syntactically and without tampering with any interpretation. We henceforth assume that $\mathrm{Var}(r) \setminus \mathrm{Var}(\ell) \subseteq \mathrm{Var}(\varphi)$ for each rewrite rule $\ell \to r\ [\varphi]$. We also say that a substitution σ respects a *logical constraint* φ if $\sigma(x)$ is a value for all $x \in \mathrm{Var}(\varphi)$ and $[\![\varphi\sigma]\!] = 1$.

Before presenting constrained HORPO, we recall that in [21] all sorts collapse into one, and for example, $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $\mathsf{int} \to \mathsf{intlist} \to \mathsf{intlist}$ are considered equal. The idea is that the original rewrite relation can be embedded in the single-sorted one, and if the latter is well-founded, so is the former. We follow this convention and henceforth compare types by their $\rightarrow$-structure only.

Below $\succ_\varphi^{\mathfrak{l}}$ and $\succ_\varphi^{\mathfrak{m}}$ are induced by $\succsim_\varphi$ and $\succ_\varphi$:

Definition 6. *Constrained HORPO depends on the following parameters:*

1. *The interpretation of theory symbols $\sqsupset_A \colon A \to A \to \mathsf{bool}$ for all $A \in \mathcal{S}_\vartheta$ such that $[\![\sqsupset_A]\!]$ is a well-founded ordering over $\mathfrak{X}_A$. The interpretation $[\![\sqsupseteq_A]\!]$ is assumed to be the reflexive closure of $[\![\sqsupset_A]\!]$. We usually write just $\sqsupset$ and $\sqsupseteq$ because sorts collapse. Consider $[\![\sqsupset]\!]$ the union $\bigcup_{A \in \mathcal{S}_\vartheta} [\![\sqsupset_A]\!]$, and $[\![\sqsupseteq]\!]$ likewise.*
2. *The* precedence $\blacktriangleright$, *a well-founded ordering over $\mathcal{F}$ such that $f \blacktriangleright g$ for all $f \in \mathcal{F} \setminus \mathcal{F}_\vartheta$ and $g \in \mathcal{F}_\vartheta$.*
3. *The* status $\mathfrak{s}$, *a mapping from $\mathcal{F}$ to $\{\mathfrak{l}, \mathfrak{m}_2, \mathfrak{m}_3, \ldots\}$.*

The higher-order recursive path ordering *(HORPO) is a family of pairs of type-preserving relations $(\succsim_\varphi, \succ_\varphi)$ indexed by logical constraints and defined by the following conditions:*

1. *$s \succsim_\varphi t$ if and only if one of the following conditions is true:*
   - *(a) s and t are theory terms whose type is a sort,* $\mathrm{Var}(s) \cup \mathrm{Var}(t) \subseteq \mathrm{Var}(\varphi)$ *and $\varphi \models s \sqsupseteq t$.*
   - *(b) $s \succ_\varphi t$.*
   - *(c) $s \downarrow_\kappa t$.*
   - *(d) s is not a theory term, $s = s_0\ s_1$, $t = t_0\ t_1$, $s_0 \succsim_\varphi t_0$ and $s_1 \succsim_\varphi t_1$.*
2. *$s \succ_\varphi t$ if and only if one of the following conditions is true:*
   - *(a) s and t are theory terms whose type is a sort,* $\mathrm{Var}(s) \cup \mathrm{Var}(t) \subseteq \mathrm{Var}(\varphi)$ *and $\varphi \models s \sqsupset t$.*
   - *(b) s and t have equal types and $s \rhd_\varphi t$.*
   - *(c) s is not a theory term, $s = f\ s_1 \cdots s_n$ for some $f \in \mathcal{F}$, $t = f\ t_1 \cdots t_n$, $\forall i.\, s_i \succsim_\varphi t_i$ and $\exists k.\, s_k \succ_\varphi t_k$.*
   - *(d) s is not a theory term, $s = x\ s_1 \cdots s_n$ for some $x \in \mathcal{V}$, $t = x\ t_1 \cdots t_n$, $\forall i.\, s_i \succsim_\varphi t_i$ and $\exists k.\, s_k \succ_\varphi t_k$.*
3. *$s \rhd_\varphi t$ if and only if s is not a theory term, $s = f\ s_1 \cdots s_m$ for some $f \in \mathcal{F}$ and one of the following conditions is true:*
   - *(a) $\exists k.\, s_k \succsim_\varphi t$.*
   - *(b) $t = t_0\ t_1 \cdots t_n$, $\forall i.\, s \rhd_\varphi t_i$.*
   - *(c) $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$, $\forall i.\, s \rhd_\varphi t_i$.*
   - *(d) $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \ldots s_m \succ_\varphi^{\mathfrak{l}} t_1 \ldots t_n$, $\forall i.\, s \rhd_\varphi t_i$.*
   - *(e) $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \ldots s_{\min(m,k)} \succ_\varphi^{\mathfrak{m}} t_1 \ldots t_k$, $\forall i.\, s \rhd_\varphi t_i$.*
   - *(f) t is a value or a variable in* $\mathrm{Var}(\varphi)$.

Here $s \downarrow_\kappa t$ if and only if there exists a term r such that $s \to_\kappa^* r$ and $t \to_\kappa^* r$.

Comparison to the Original HORPO. Conditions 1d, 2c and 2d are included in the definition so that $\succsim_\varphi$ and $\succ_\varphi$ are rule-monotonic. We stress that it is mandatory to use the weakened, rule-monotonicity requirement instead of the traditional monotonicity requirement: if $\succ_\mathfrak{t}$ is monotonic, $1 \succ_\mathfrak{t} 0$ implies $1 - 1 \succ_\mathfrak{t} 1 - 0$, but $\mathfrak{t} \models (1 - 0) \sqsupset (1 - 1)$, i.e., $\succ_\mathfrak{t}$ cannot possibly be well-founded.

From curried notation, another issue related to rule-monotonicity arises, which leads to the above definition of $\rhd_\varphi$. If we had the original HORPO naively mirrored, the definition of $\succ_\varphi$ would include a condition which corresponds to condition 3b and reads: "$s \succ_\varphi t$ if s is not a theory term, $s = f\ s_1 \cdots s_m$ for some $f \in \mathcal{F}$, $t = t_0\ t_1 \cdots t_n$ and $\forall i.\, s \succ_\varphi t_i \vee \exists k.\, s_k \succsim_\varphi t_i$". Assume given such terms s and t, and that, say, $s \succ_\varphi t_1$. Now if there is a term r to which s can be applied, we have a problem with proving $s\ r \succ_\varphi t\ r = t_0\ t_1 \cdots t_n\ r$ because $s\ r \succ_\varphi t_1$ is not obtainable due to the type restriction. Note that $\succsim_\varphi$ and $\succ_\varphi$ are by definition type-preserving, whereas $\rhd_\varphi$ is not.

This limitation is overcome by means of $\rhd_\varphi$, which actually makes the overall definition more powerful, and is reminiscent of the distinction between $\succ$ and $\succ_{\mathcal{T}_S}$ in later versions of HORPO (e.g., [5]). Other extensions from these works, however, are not yet included in the above definition, and except for the type restriction and uncurried notation, the conditions of $\rhd_\varphi$ largely match those of the original HORPO.

Another subtle difference is the use of *generalized* lexicographic and multiset orderings: in the original HORPO, $\succsim$ is the reflexive closure of $\succ$, and therefore it suffices to use the more traditional definitions of lexicographic and multiset orderings. Here, as observed above, this would be unnecessarily restrictive.

The split of a single multiset status label in $\mathfrak{m}_2, \mathfrak{m}_3, \ldots$ is due to curried notation—in particular, the possibility of partial application. If we had only a single multiset status label, which would admit, for example, both $f\ 2\ 2 \rhd_\mathfrak{t} f\ 1$ and $f\ 1\ 3 \succ_\mathfrak{t} f\ 2\ 2$, it would be possible that $\succ_\mathfrak{t}$ is not well-founded: note that $g\ (f\ 1) \succ_\mathfrak{t} f\ 1\ 3$ due to, among others, conditions 2b and 3b, and if $f \blacktriangleright g$, we would then have $f\ 2\ 2 \succ_\mathfrak{t} g\ (f\ 1)$ due to, among others, conditions 2b and 3c. This change adds some power to constrained HORPO: we can prove, for example, the termination of the single-rule system $\mathsf{f}\ x\ \mathsf{a}\ y \to \mathsf{f}\ \mathsf{b}\ x\ (\mathsf{g}\ y)$ by choosing $\mathfrak{s}(\mathsf{f}) = \mathfrak{m}_2$, which is not possible if all arguments must be considered, as the original HORPO requires. We do not need $\mathfrak{m}_1$ because this case is already covered by choosing $\mathfrak{l}$.

Given an LCSTRS $\mathcal{R}$, if we can divide the set of rules into two subsets $\mathcal{R}_1$ and $\mathcal{R}_2$, and find a combination of $[\![\sqsupset]\!]$, $\blacktriangleright$ and $\mathfrak{s}$ that guarantees $\ell \succsim_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}_1$ and $\ell \succ_\varphi r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}_2$, the termination of $\mathcal{R}$ is reduced to that of $\mathcal{R}_1$. Before proving the soundness, we check out some examples:

Example 9. We continue the analysis of the motivating example rec. Let $[\![\sqsupset_{\mathsf{int}}]\!]$ be $\lambda m.\, \lambda n.\, m > 0 \wedge m > n$ as above. There is only one function symbol in $\mathcal{F} \setminus \mathcal{F}_\vartheta$, and it turns out that $\blacktriangleright$ can be any precedence. Let $\mathfrak{s}$ be a mapping such that $\mathfrak{s}(\mathsf{rec}) = \mathfrak{l}$. The first rewrite rule can be removed due to conditions 2b and 3a. The second rewrite rule can be removed as follows:

1. $\mathsf{rec}\ n\ x\ f \succ_{n>0} f\ (n - 1)\ (\mathsf{rec}\ (n - 1)\ x\ f)$ by 2b, 2.

2. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} f\ (n-1)\ (\mathsf{rec}\ (n-1)\ x\ f)$ by 3b, 3, 4, 5.
3. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} f$ by 3a, 6.
4. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} n-1$ by 3a, 7.
5. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} \mathsf{rec}\ (n-1)\ x\ f$ by 3d, 8, 4, 9, 3.
6. $f \succsim_{n>0} f$ by 1c.
7. $n \succsim_{n>0} n-1$ by 1a.
8. $n \succ_{n>0} n-1$ by 2a.
9. $\mathsf{rec}\ n\ x\ f \rhd_{n>0} x$ by 3a, 10.
10. $x \succsim_{n>0} x$ by 1c.

Example 10. Consider Example 5. Let $[\![\sqsupset_{\mathsf{int}}]\!]$ be $\lambda m.\,\lambda n.\,m > 0 \wedge m > n$. Let $\blacktriangleright$ be a precedence such that $\mathsf{take} \blacktriangleright \mathsf{nil}$ and $\mathsf{take} \blacktriangleright \mathsf{cons}$. Let $\mathfrak{s}$ be a mapping such that $\mathfrak{s}(\mathsf{take}) = \mathfrak{l}$. Then we can remove all of the rewrite rules. Note that to establish $\mathsf{take}\ n\ (\mathsf{cons}\ x\ l) \succ_{n>0} \mathsf{cons}\ x\ (\mathsf{take}\ (n-1)\ l)$, we need $\mathsf{cons}\ x\ l \succsim_{n>0} x$, which is obtainable because $\mathsf{intlist}$ is not distinguished from int.

4.4 Properties of Constrained HORPO

The soundness of constrained HORPO as a technique for rule removal relies on the following properties, which we now prove.

Rule Orientation. The goal consists of two parts: $\succsim_{\mathfrak{t}}$ stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succsim_{\varphi} r$, and $\succ_{\mathfrak{t}}$ stably orients a rewrite rule $\ell \to r\ [\varphi]$ if $\ell \succ_{\varphi} r$. The core of the argument is to prove the following lemma:

Lemma 2. *Given logical constraints φ and φ' such that* $\mathrm{Var}(\varphi) \supseteq \mathrm{Var}(\varphi')$ *and $\varphi \models \varphi'$, $\mathfrak{t} \models \varphi'\sigma$ holds for each substitution σ which respects φ.*

Proof. It follows from $\varphi \models \varphi'$ that $[\![\varphi'\sigma]\!] = 1$. Note that $\mathrm{Var}(\varphi'\sigma) = \emptyset$, and therefore $\varphi'\sigma\sigma' = \varphi'\sigma$ for all σ'. Hence, $\mathfrak{t} \models \varphi'\sigma$. □

And the rest is routine:

Theorem 2. *Given a logical constraint φ, terms s and t, the following statements are true for each substitution σ which respects φ:*

1. *$s \succsim_{\varphi} t$ implies $s\sigma \succsim_{\mathfrak{t}} t\sigma$.*
2. *$s \succ_{\varphi} t$ implies $s\sigma \succ_{\mathfrak{t}} t\sigma$.*
3. *$s \rhd_{\varphi} t$ implies $s\sigma \rhd_{\mathfrak{t}} t\sigma$.*

Proof. By mutual induction on the derivation. Note that $\to_{\kappa}$ is stable. □

Rule-Monotonicity. Both $\succsim_{\varphi}$ and $\succ_{\varphi}$ are rule-monotonic for all φ. The former is trivial to prove, and the key to proving the latter is the following lemma:

Lemma 3. *$f\ s_1 \cdots s_m\ r \rhd_{\varphi} t$ if $f\ s_1 \cdots s_m \rhd_{\varphi} t$.*

Proof. By induction on the derivation. □

Now we can prove the rule-monotonicity:

Theorem 3. $\succ_\varphi$ *is rule-monotonic.*

Proof. By induction on the context $C[]$. Essentially, we ought to prove that given terms s and t which have equal types, if s is not a theory term and $s \succ_\varphi t$, $s\ r \succ_\varphi t\ r$ for all r, and $r\ s \succ_\varphi r\ t$ for all r. We prove the former by case analysis on the derivation of $s \succ_\varphi t$, and prove the latter by case analysis on r: $r = f\ r_1 \cdots r_n$ for some $f \in \mathcal{F}$ or $r = x\ r_1 \cdots r_n$ for some $x \in \mathcal{V}$. □

Compatibility. The strict relation $\succ_\mathfrak{t}$ is compatible with its non-strict counterpart $\succsim_\mathfrak{t}$; we prove that $\succsim_\mathfrak{t} ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup (\succ_\mathfrak{t} ; \succ_\mathfrak{t})$, given the following observation:

Theorem 4. $\succsim_\mathfrak{t} = \succ_\mathfrak{t} \cup \downarrow_\kappa$.

Proof. By definition, $\succsim_\mathfrak{t} \supseteq \succ_\mathfrak{t} \cup \downarrow_\kappa$. We prove $\succsim_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup \downarrow_\kappa$ by induction on the derivation of $s \succsim_\mathfrak{t} t$. Only two cases are non-trivial. If s and t are ground theory terms whose type is a sort and $[\![s \sqsupseteq t]\!] = 1$, we have either $[\![s \sqsupset t]\!] = 1$ or $[\![s]\!] = [\![t]\!]$, and the former implies $s \succ_\mathfrak{t} t$ while the latter implies $s \downarrow_\kappa t$. On the other hand, if s is not a theory term, $s = s_0\ s_1$, $t = t_0\ t_1$, $s_0 \succsim_\mathfrak{t} t_0$ and $s_1 \succsim_\mathfrak{t} t_1$, by induction, if $s_0 \succ_\mathfrak{t} t_0$ or $s_1 \succ_\mathfrak{t} t_1$, we can prove $s \succ_\mathfrak{t} t$ in the same manner as we prove the rule-monotonicity of $\succ_\mathfrak{t}$, or $s_0 \downarrow_\kappa t_0$ and $s_1 \downarrow_\kappa t_1$, then $s \downarrow_\kappa t$. □

Theorem 4 plays an important role in the well-foundedness proof of $\succ_\mathfrak{t}$ as well.

For the compatibility of $\succ_\mathfrak{t}$ with $\succsim_\mathfrak{t}$, it remains to prove that $\downarrow_\kappa ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t}$, which is implied by the following lemma:

Lemma 4. *Given terms s and s' such that $s \to_\kappa s'$, the following statements are true for all t:*

1. *$s \succsim_\mathfrak{t} t$ if and only if $s' \succsim_\mathfrak{t} t$.*
2. *$s \succ_\mathfrak{t} t$ if and only if $s' \succ_\mathfrak{t} t$.*
3. *$s \rhd_\mathfrak{t} t$ if and only if $s' \rhd_\mathfrak{t} t$.*

Proof. By mutual induction on the derivation for "if" and "only if" separately. Note that $\to_\kappa$ is confluent. □

The compatibility follows as a corollary:

Corollary 1. $\succsim_\mathfrak{t} ; \succ_\mathfrak{t} \subseteq \succ_\mathfrak{t} \cup (\succ_\mathfrak{t} ; \succ_\mathfrak{t})$.

Well-Foundedness. Following [21], we base the well-foundedness proof of $\succ_\mathfrak{t}$ on the predicate of computability [40,17]. There are, however, two major differences, which pose new technical challenges: $\succsim_\mathfrak{t}$ is no more the reflexive closure of $\succ_\mathfrak{t}$ and curried notation instead of uncurried notation is in use.

In Definition 6, $\succ_\varphi^\mathfrak{l}$ and $\succ_\varphi^\mathfrak{m}$ are induced by $\succsim_\varphi$ and $\succ_\varphi$. We need certain properties of $\succ_\mathfrak{t}^\mathfrak{l}$ and $\succ_\mathfrak{t}^\mathfrak{m}$ to prove that $\succ_\mathfrak{t}$ is well-founded. Because $\succsim_\mathfrak{t}$ is neither the equality over terms nor the reflexive closure of $\succ_\mathfrak{t}$, those properties are less standard and deserve inspection. The property of $\succ_\mathfrak{t}^\mathfrak{l}$ is relatively easy to prove:

Theorem 5. *Given relations $\succsim$ and $\succ$ over X such that $\succ$ is well-founded and $\succsim$; $\succ \subseteq \succ^+$, $\succ^{\mathfrak{l}}$ is well-founded over X^n for all n.*

Proof. The standard method used when $\succsim$ is the equality still applies. □

We refer to [41] for the proof of the following property of $\succ_{\mathfrak{t}}^{\mathfrak{m}}$:

Theorem 6. *Given relations $\succsim$ and $\succ$ over X such that $\succsim$ is a quasi-ordering, $\succ$ is well-founded and $\succsim$; $\succ \subseteq \succ$, $\succ^{\mathfrak{m}}$ is well-founded over X^*.*

Proof. See Theorem 3.7 in [41]. □

In comparison to [41], we waive the transitivity requirement for $\succ$ above, but we cannot get around the requirement that $\succsim$ is a quasi-ordering without significantly changing the proof. This seems problematic because $\succsim_{\mathfrak{t}}$ is not necessarily transitive due to its inclusion of $\succ_{\mathfrak{t}}$. Fortunately, one observation resolves this issue: $\succ_{\mathfrak{t}}^{\mathfrak{m}}$ can equivalently be seen as induced by $\downarrow_\kappa$ and $\succ_{\mathfrak{t}}$ due to Theorem 4. In the same spirit, we can prove the following property:

Theorem 7. *$\downarrow_\kappa^{\mathfrak{m}}$; $\succ_{\mathfrak{t}}^{\mathfrak{m}} \subseteq \succ_{\mathfrak{t}}^{\mathfrak{m}}$ where $s_1 \dots s_n \downarrow_\kappa^{\mathfrak{m}} t_1 \dots t_n$ if and only if there exists a permutation π over $\{1, \dots, n\}$ such that $s_{\pi(i)} \downarrow_\kappa t_i$ for all i.*

Proof. See Lemma 3.2 in [41]. □

Our definition of computability (or reducibility [17]) is standard:

Definition 7. *A term t_0 is called* computable *if either*

1. *the type of t_0 is a sort and t_0 is terminating with respect to $\succ_{\mathfrak{t}}$, or*
2. *the type of t_0 is $A \to B$ and $t_0\ t_1$ is computable for all computable $t_1 : A$.*

In [21], a term is called neutral if it is not a λ-abstraction. Due to the exclusion of λ-abstractions, one might consider all LCSTRS terms neutral. This naive definition, however, does not capture the essence of neutrality: if a term t_0 is neutral, a one-step reduct (with respect to $\succ_{\mathfrak{t}}$) of $t_0\ t_1$ can only be $t_0'\ t_1'$ where t_0' and t_1' are reducts of t_0 and t_1, respectively. Because of curried notation, neutral LCSTRS terms should be defined as follows:

Definition 8. *A term is called* neutral *if it assumes the form $x\ t_1 \cdots t_n$ for some variable x.*

And we recall the following results:

Theorem 8. *Computable terms have the following properties:*

1. *Given terms s and t such that $s \succ_{\mathfrak{t}} t$, if s is computable, so is t.*
2. *All computable terms are terminating with respect to $\succ_{\mathfrak{t}}$.*
3. *Given a neutral term s, if t is computable for all t such that $s \succ_{\mathfrak{t}} t$, so is s.*

Proof. The standard proof still works despite the seemingly different definition of neutrality. □

In addition, we prove the following lemma:

Lemma 5. *Given terms s and t such that $s \downarrow_\kappa t$, if s is computable, so is t.*

Proof. By induction on the type of s and t. □

And we have the following corollary due to Theorem 4:

Corollary 2. *Given terms s and t such that $s \succsim_{\mathfrak{t}} t$, if s is computable, so is t.*

The goal is to prove that all terms are computable. To do so, the key is to prove that $f\ s_1 \cdots s_m$ is computable where f is a function symbol if s_i is computable for all i. In [21], this is done on the basis that $f\ s_1 \cdots s_m$ is neutral, which is not true in our case. We do it differently and start with a definition:

Definition 9. *Given $f : A_1 \to \cdots \to A_n \to B$ where $f \in \mathcal{F}$ and $B \in \mathcal{S}$, let $\mathrm{ar}(f)$ be n. We introduce a special symbol $\top$ and extend our previous definitions so that $\top \succ_{\mathfrak{t}} t$ for all $t \in T(\mathcal{F}, \mathcal{V})$ and $\top \downarrow_\kappa \top$. This way $\top \succsim_{\mathfrak{t}} t$ if $t \in T(\mathcal{F}, \mathcal{V})$ or $t = \top$. Given terms $\bar{t} = t_1 \ldots t_n$, let $(\bar{t})_k$ be t_k if $k \leq n$, and $\top$ if $k > n$. Given terms $s = f\ s_1 \cdots s_m$ and $t = g\ t_1 \cdots t_n$ where $f \in \mathcal{F}$, $g \in \mathcal{F}$, all s_i and t_i are computable, we define $\succ_c$ such that $s \succ_c t$ if and only if $f \blacktriangleright g$, or $f = g$ and*

- *$\mathfrak{s}(f) = \mathfrak{l}$ and $(\bar{s})_1 \ldots (\bar{s})_{\mathrm{ar}(f)} \succ_{\mathfrak{t}}^{\mathfrak{l}} (\bar{t})_1 \ldots (\bar{t})_{\mathrm{ar}(f)}$, or*
- *$\mathfrak{s}(f) = \mathfrak{m}_k$ and*
  - *$(\bar{s})_1 \ldots (\bar{s})_k \succ_{\mathfrak{t}}^{\mathfrak{m}} (\bar{t})_1 \ldots (\bar{t})_k$, or*
  - *$(\bar{s})_1 \ldots (\bar{s})_k \downarrow_\kappa^{\mathfrak{m}} (\bar{t})_1 \ldots (\bar{t})_k$, $\forall i > k.\, (\bar{s})_i \succsim_{\mathfrak{t}} (\bar{t})_i$ and $\exists i > k.\, (\bar{s})_i \succ_{\mathfrak{t}} (\bar{t})_i$.*

This gives us a well-founded relation:

Lemma 6. *$\succ_c$ is well-founded.*

Proof. Since all computable terms are terminating with respect to $\succ_{\mathfrak{t}}$, $\succ_{\mathfrak{t}}$ is well-founded over computable terms. The introduction of $\top$ clearly does not break this well-foundedness. The outermost layer of $\succ_c$ regards $\blacktriangleright$, which is well-founded by definition. We need only to fix the function symbol f and to go deeper. If $\mathfrak{s}(f) = \mathfrak{l}$, we know that $\succ_{\mathfrak{t}}^{\mathfrak{l}}$ is well-founded over lists of length $\mathrm{ar}(f)$ because of Theorem 5. If $\mathfrak{s}(f) = \mathfrak{m}_k$, $\succ_c$ splits each list of arguments in two and performs a lexicographic comparison. We can go past the first component because of Theorems 6 and 7. And the rest, a pointwise comparison, is also well-founded. So we can conclude that $\succ_c$ is well-founded. □

Now we prove the aforementioned statement:

Lemma 7. *Given a term $s = f\ s_1 \cdots s_m$ where f is a function symbol, if s_i is computable for all i, so is s.*

Proof. By well-founded induction on $\succ_c$. We consider the type of s:

- If the type is a sort, we ought to prove that s is terminating with respect to $\succ_{\mathfrak{t}}$. We need only to consider the cases in which s is not a theory term because all theory terms are terminating with respect to $\succ_{\mathfrak{t}}$ due to the well-foundedness of $[\![\sqsupset]\!]$. Take an arbitrary term t such that $s \succ_{\mathfrak{t}} t$. We prove that t is terminating with respect to $\succ_{\mathfrak{t}}$ by case analysis on the derivation of $s \succ_{\mathfrak{t}} t$. If $t = f\ t_1 \cdots t_m$, $\forall i.\, s_i \succsim_{\mathfrak{t}} t_i$ and $\exists k.\, s_k \succ_{\mathfrak{t}} t_k$, we can prove that $s \succ_c t$. By induction, t is computable and therefore terminating with respect to $\succ_{\mathfrak{t}}$. If $s \rhd_{\mathfrak{t}} t$, we prove that t is computable for all t such that $s \rhd_{\mathfrak{t}} t$ (t is generalized) by inner induction on the derivation of $s \rhd_{\mathfrak{t}} t$:
  1. If $\exists k.\, s_k \succsim_{\mathfrak{t}} t$, t is computable due to Corollary 2.
  2. If $t = t_0\ t_1 \cdots t_n$ and $\forall i.\, s \rhd_{\mathfrak{t}} t_i$, t_i is computable for all i by inner induction. By definition, t is computable.
  3. If $t = g\ t_1 \cdots t_n$, $f \blacktriangleright g$ and $\forall i.\, s \rhd_{\mathfrak{t}} t_i$, t_i is computable for all i by inner induction. It follows from $f \blacktriangleright g$ that $s \succ_c t$, and t is computable by outer induction.
  4. If $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{l}$, $s_1 \dots s_m \succ_{\mathfrak{t}}^{\mathfrak{l}} t_1 \dots t_n$ and $\forall i.\, s \rhd_{\mathfrak{t}} t_i$, t_i is computable for all i by inner induction. Likewise, $s \succ_c t$.
  5. If $t = f\ t_1 \cdots t_n$, $\mathfrak{s}(f) = \mathfrak{m}_k$, $k \leq n$, $s_1 \dots s_{\min(m,k)} \succ_{\mathfrak{t}}^{\mathfrak{m}} t_1 \dots t_k$ and $\forall i.\, s \rhd_{\mathfrak{t}} t_i$, t_i is computable for all i by inner induction. Likewise, $s \succ_c t$.
  6. If t is a value, t is terminating with respect to $\succ_{\mathfrak{t}}$ and its type is a sort.
- If the type is $A \to B$, take an arbitrary computable $s_{m+1} : A$. We prove that $s \succ_c s\ s_{m+1} = f\ s_1 \cdots s_{m+1}$. Note that $(s_1 \dots s_m)_i = (s_1 \dots s_{m+1})_i$ for all $i \leq m$ and $(s_1 \dots s_m)_{m+1} = \top \succ_{\mathfrak{t}} (s_1 \dots s_{m+1})_{m+1}$. Consider $\mathfrak{s}(f) = \mathfrak{l}$, $\mathfrak{s}(f) = \mathfrak{m}_k$ while $k > m$, and $\mathfrak{s}(f) = \mathfrak{m}_k$ while $k \leq m$. We have $s \succ_c s\ s_{m+1}$ in each case. By induction, $s\ s_{m+1}$ is computable. Hence, s is computable.

We conclude that s is computable. □

Now the well-foundedness of $\succ_{\mathfrak{t}}$ follows immediately:

Theorem 9. $\succ_{\mathfrak{t}}$ *is well-founded.*

Proof. We prove that every term t is computable by induction on t. Given Lemma 7, we need only to prove that variables are computable, which is the case because variables are neutral and in normal form with respect to $\succ_{\mathfrak{t}}$. □

5 Discussion: HORPO and Dependency Pairs

In Section 4, we discussed rule removal, and presented a reduction ordering to prove termination. However, in practice it is not so common to directly use reduction orderings as a termination method. Rather, the norm in the literature nowadays is to use dependency pairs.

The dependency pair framework [1,16] allows a single term rewriting system to be split into multiple "DP problems", each of which can then be analyzed independently. The framework operates by iteratively simplifying DP problems until none remain, in which case the original system is proved terminating. There are variants for many styles of term rewriting, including first-order LCTRSs [25] and unconstrained higher-order TRSs [39,25,11].

Importantly, many existing techniques can be reformulated as "processors" (DP problem simplifiers) in the dependency pair framework. Such techniques include reduction orderings, which are at the heart of the dependency pair framework. This combination is far more powerful than using reduction orderings directly because the monotonicity requirement is replaced by weak monotonicity, and we do not have to orient the entire system in one go.

Consider the following first-order LCTRS:

$$\mathsf{u}\ x\ y \to \mathsf{u}\ (x+1)\ (y*2) \quad [x < 100] \qquad \mathsf{v}\ y \to \mathsf{v}\ (y-1) \quad [y > 0]$$
$$\mathsf{u}\ 100\ y \to \mathsf{v}\ y$$

This system cannot be handled by HORPO directly: the ordering $[\![\sqsupset_{\mathsf{int}}]\!]$ needs to be fixed globally, so we can either orient the rewrite rule at the top-left corner or the one at the top-right corner, but not both at the same time. We could address this dilemma by using a more elaborate definition of HORPO (for example, by giving every function symbol an additional status that indicates the theory ordering to be used for each of its arguments), but this seems redundant: in practice, such a system would be handled by the dependency pair framework. Following the definition in [25], the above system would be split in two separate DP problems corresponding to the two loops:

$$\{\, \mathsf{u}^\sharp\ x\ y \to \mathsf{u}^\sharp\ (x+1)\ (y*2) \quad [x < 100]\,\} \qquad \{\, \mathsf{v}^\sharp\ y \to \mathsf{v}^\sharp\ (y-1) \quad [y > 0]\,\}$$

which could then be handled independently.

While dependency pairs for LCSTRSs are not yet defined (and beyond the scope of this paper), we postulate that the definitions for curried higher-order rewriting in [11] and first-order constrained rewriting in [25] can be combined in a natural way. In this setting, HORPO would naturally be combined with *argument filterings* [1,11]. That is, since we only require *weak* monotonicity, some arguments can be removed. For example, the first DP problem above can be handled by showing the following inequalities:

$$\mathsf{u}^\sharp\ x \succ_{x<100} \mathsf{u}^\sharp\ (x+1) \qquad \mathsf{u} \succsim_{x<100} \mathsf{u} \qquad \mathsf{v} \succsim_{y>0} \mathsf{v} \qquad \mathsf{u} \succsim_{\mathfrak{t}} \mathsf{v}$$

This is the case with $\mathsf{u} \blacktriangleright \mathsf{v}$.

6 Implementation

A preliminary implementation of LCSTRSs is available in Cora through the link:

https://github.com/hezzel/cora

Cora is an open-source analyzer for constrained rewriting, which can be used both as a stand-alone tool and as a library. Note that Cora is still in active development, and its functionalities, as well as its interface, are subject to change.

Nevertheless, Cora is already used in several student projects. Cora supports only the theories of integers and booleans so far, but is intended to eventually support any theory, provided that an SMT solver is able to handle it. Example input files are supplied in the above repository. The version of this paper is available in [28].

Automating Constrained HORPO. Cora includes an implementation of constrained HORPO. Following existing termination tools such as AProVE [14], NaTT [42] and Wanda [26], we use an SMT encoding such that a satisfying assignment to variables in the SMT problem corresponds to a combination of the precedence $\blacktriangleright$, the status $\mathfrak{s}$ and the ordering $[\![\sqsupset_{\mathsf{int}}]\!]$ that proves the termination of the encoded system by constrained HORPO. As for booleans, we simply choose the ordering $[\![\sqsupset_{\mathsf{bool}}]\!]$ such that $[\![\mathfrak{t} \sqsupset_{\mathsf{bool}} \mathfrak{f}]\!] = 1$.

To encode the precedence and the status, we introduce integer variables $\mathtt{prec}_f$ and $\mathtt{stat}_f$ for each function symbol f that is not a value. We require that $\mathtt{prec}_f < 0$ if f is a theory symbol, and that $\mathtt{prec}_f \geq 0$ otherwise—so that $\mathtt{prec}_f > \mathtt{prec}_g$ corresponds to $f \blacktriangleright g$. The value k of $\mathtt{stat}_f$ indicates $\mathfrak{s}(f) = \mathfrak{l}$ if $k = 1$, and $\mathfrak{s}(f) = \mathfrak{m}_k$ if $k > 1$. We let $\mathtt{down}$ be a boolean variable which indicates the choice between two possibilities for $[\![\sqsupset_{\mathsf{int}}]\!]$: $\lambda m. \lambda n. m > -M \wedge m > n$ and $\lambda m. \lambda n. m < M \wedge m < n$ (the choice of M is discussed below).

In the derivation of $s \succ_\varphi t$, all assertions assume the form $s' \; R_\varphi \; t'$ where s' and t' are subterms of s and t, respectively (see Example 9). Hence, given a finite set of rewrite rules, there are only finitely many possible assertions to be analyzed. By inspecting the definition of constrained HORPO, we also note that there are no cyclic dependencies. For all $\ell \to r \; [\varphi]$, respective subterms s and t of ℓ and r, and $R \in \{ \succsim, \succ, \rhd, 1\mathrm{a}, 1\mathrm{b}, \dots, 3\mathrm{f} \}$, we thus introduce a variable $\langle s \; R_\varphi \; t \rangle$ with its defining constraint. Without going into detail for all the cases, we provide a few key examples:

– If s and t do not have equal types, we add $\neg \langle s \succsim_\varphi t \rangle$; otherwise, we add $\langle s \succsim_\varphi t \rangle \implies \langle s \; 1\mathrm{a}_\varphi \; t \rangle \vee \langle s \; 1\mathrm{b}_\varphi \; t \rangle \vee \langle s \; 1\mathrm{c}_\varphi \; t \rangle \vee \langle s \; 1\mathrm{d}_\varphi \; t \rangle$, which states that if $s \succsim_\varphi t$ holds, it must hold in one of the defining cases 1a, 1b, 1c and 1d. Each of these cases in turn has its defining constraint.
– $\langle f \; s_1 \cdots s_m \; 3\mathrm{c}_\varphi \; g \; t_1 \cdots t_n \rangle \implies \mathtt{prec}_f > \mathtt{prec}_g \wedge \bigwedge_j \langle f \; s_1 \cdots s_m \rhd_\varphi t_j \rangle$.
– We come up with the defining constraint of $\langle s \; 2\mathrm{a}_\varphi \; t \rangle$ by case analysis:
  • If either of s and t is not a theory term, or their respective types are not the same theory sort, or $\mathrm{Var}(s) \cup \mathrm{Var}(t) \nsubseteq \mathrm{Var}(\varphi)$, we add $\neg \langle s \; 2\mathrm{a}_\varphi \; t \rangle$.
  • Otherwise, we consider the type of s and t:
    ∗ The type is int. We respectively check if $\varphi \implies s > -M \wedge s > t$ and $\varphi \implies s < M \wedge s < t$ are valid. If the former is *not* valid, we add $\langle s \; 2\mathrm{a}_\varphi \; t \rangle \implies \neg\mathtt{down}$; if the latter is *not* valid, we add $\langle s \; 2\mathrm{a}_\varphi \; t \rangle \implies \mathtt{down}$. That is, if both of the validity checks fail, both of the constraints are added, which is equivalent to adding $\neg \langle s \; 2\mathrm{a}_\varphi \; t \rangle$.
    ∗ The type is bool. We add $\neg \langle s \; 2\mathrm{a}_\varphi \; t \rangle$ if $\varphi \implies s \wedge \neg t$ is not valid; if it is valid, nothing is added and the SMT solver is free to set true for the variable $\langle s \; 2\mathrm{a}_\varphi \; t \rangle$.

Here M is twice the largest absolute value of integers occurring in the rewrite rules, or just 1000 if that is too large—this value is chosen arbitrarily. Note that the validity checks are *not* included as part of the SMT problem: if they were included, the satisfiability problem would contain universal quantification, which is typically hard to solve. We rather pose a separate question to the SMT solver every time we encounter theory comparison, and for integers, consider whether the pair can be oriented downward with $\lambda m.\, \lambda n.\, m > -M \wedge m > n$, upward with $\lambda m.\, \lambda n.\, m < M \wedge m < n$, or not at all. Hence, we must fix the bound M beforehand.

– The hardest is 3e: we need not only to encode the multiset comparison, but also to make sure that only k arguments are to be considered on both sides (should there be more). Following Definition 4, we introduce boolean variables $\mathtt{strict}_1, \ldots, \mathtt{strict}_m$ where $\mathtt{strict}_i$ indicates $i \in I$, and integer variables $\pi(1), \ldots, \pi(n)$. The defining constraint of $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle$ is the conjunction of the following components:
  - $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies 2 \le \mathtt{stat}_f \le n$.
  - $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \bigwedge_j \langle f\ s_1 \cdots s_m \rhd_\varphi t_j \rangle$.
  - $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \bigvee_i \mathtt{strict}_i$.
  - For all $i \in \{1, \ldots, m\}$, $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \mathtt{strict}_i \implies i \le \mathtt{stat}_f$. That is, $I \subseteq \{1, \ldots, k\}$ if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $j \in \{1, \ldots, n\}$, $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge j \le \mathtt{stat}_f \implies 1 \le \pi(j) \wedge \pi(j) \le m \wedge \pi(j) \le \mathtt{stat}_f$. That is, $1 \le \pi(j) \le \min(m, k)$ for all $j \in \{1, \ldots, k\}$ if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n-1\}$ and $j' \in \{j+1, \ldots, n\}$, $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \implies \mathtt{strict}_i \vee \pi(j) \ne i \vee \pi(j') \ne i$. That is, $\left|\pi^{-1}(i)\right| \le 1$ for all $i \in \{1, \ldots, m\} \setminus I$—which suffices because we can add to I all $i \in \{1, \ldots, \min(m, k)\} \setminus I$ such that $\left|\pi^{-1}(i)\right| = 0$ without changing the generalized multiset ordering if $\mathfrak{s}(f) = \mathfrak{m}_k$.
  - For all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \pi(j) = i \wedge \mathtt{strict}_i \implies \langle s_i \succ_\varphi t_j \rangle$.
  - For all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, $\langle f\ s_1 \cdots s_m\ \mathrm{3e}_\varphi\ f\ t_1 \cdots t_n \rangle \wedge \pi(j) = i \wedge \neg\mathtt{strict}_i \implies \langle s_i \succsim_\varphi t_j \rangle$.

Cora succeeds in proving that all the examples in this paper are terminating, except Example 2, which is non-terminating.

7 Related Work

In this section, we assess the newly proposed formalism and the prospects for its application by comparing and relating it to the literature.

Term Rewriting. The closest related work is LCTRSs [27,12], the first-order formalism for constrained rewriting upon which the present work is built. Similarly, there are numerous formalisms for higher-order term rewriting, but without built-in logical constraints, e.g., [21,22,31]. It seems likely that the methods for analyzing those can be extended with support for SMT, as what is done for HORPO in this paper.

Also worth mentioning is the K Framework [35], which, like our formalism, can be used as an intermediate language for program analysis and is based on a form of first-order rewriting. The K tool includes techniques through *reachability logic*, rather than methods like HORPO.

There are several works that analyze functional programs using term rewriting, e.g., [2,15]. However, they typically use translations to first-order systems. Hence, some of the structure of the initial problem is lost, and their power is weakened.

HORPO. Our definition of constrained HORPO is based on the first-order constrained RPO for LCTRSs [27] and the first definition of higher-order RPO [21]. There have been other HORPO extensions since, e.g., [5,6], and we believe that the ideas for these extensions can also be applied to constrained HORPO. We have not done so because the purpose of this paper is to show *that* and *how* techniques for analyzing higher-order systems extend, not to introduce the most powerful (and consequently more elaborate) ones.

Also worth mentioning is [4], a higher-order RPO for λ-free systems. This variant is defined for the purpose of superposition rather than termination analysis, and is ground-total but generally not monotonic.

Functional Programming. There are many works performing direct analyses of functional programs, including termination analysis, although they typically concern specific programming languages such as Haskel (e.g., [19]) and OCaml (e.g., [20]). A variety of techniques have been proposed, such as sized types [33] and decreasing measures on data [18], but as far as we can find, there is no real parallel of many rewriting techniques such as RPO. We hope that, through LCSTRSs, we can help make the techniques of term rewriting available to the functional programming community.

8 Conclusion and Future Work

In summary, we have defined a higher-order extension of logically constrained term rewriting systems, which can represent realistic higher-order programs in a natural way. To illustrate how such systems may be analyzed, we have adapted HORPO, one of the oldest higher-order termination techniques, to handle logical constraints. Despite being a very basic method, this is already powerful enough to handle examples in this paper. Both LCSTRSs and constrained HORPO are implemented in our new analysis tool Cora.

In the future, we intend to extend more techniques, both first-order and higher-order, to this formalism, and to implement them in a fully automatic tool. We hope that this will make the methods of the term rewriting community available to other communities, both by providing a powerful backend tool, and by showing how existing techniques can be adapted—so they may also be natively adopted in program analysis.

A natural starting point is to increase our power in termination analysis by extending dependency pairs [1,39,11,25] and various supporting methods like the subterm criterion and usable rules. In addition, methods for analyzing complexity, reachability and equivalence (e.g., through rewriting induction [34,12]), which have been defined for first-order LCTRSs, are natural directions for higher-order extension as well.

Acknowledgments. The authors are supported by NWO VI.Vidi.193.075, project "CHORPE". We thank Deivid Vale for his work on Cora and his assistance in the preparation of the artifact, Carsten Fuhs for his comments on an early draft of this paper, and the anonymous reviewers for their helpful feedback.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this paper.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS **236**(1–2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
2. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: Reppy, J. (ed.) Proc. ICFP. pp. 152–164 (2015). https://doi.org/10.1145/2784731.2784753
3. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). https://smtlib.cs.uiowa.edu
4. Blanchette, J.C., Waldmann, U., Wand, D.: A lambda-free higher-order recursive path order. In: Esparza, J., Murawski, A.S. (eds.) Proc. FoSSaCS. pp. 461–479 (2017). https://doi.org/10.1007/978-3-662-54458-7_27
5. Blanqui, F., Jouannaud, J.P., Rubio, A.: HORPO with computability closure: a reconstruction. In: Dershowitz, N., Voronkov, A. (eds.) Proc. LPAR. pp. 138–150 (2007). https://doi.org/10.1007/978-3-540-75560-9_12
6. Blanqui, F., Jouannaud, J.P., Rubio, A.: The computability path ordering: the end of a quest. In: Kaminski, M., Martini, S. (eds.) Proc. CSL. pp. 1–14 (2008). https://doi.org/10.1007/978-3-540-87531-4_1
7. Ciobâcă, Ş., Lucanu, D., Buruiană, A.S.: Operationally-based program equivalence proofs using LCTRSs. JLAMP **135**, 100894:1–100894:22 (2023). https://doi.org/10.1016/j.jlamp.2023.100894
8. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt, R.A. (ed.) Proc. CADE. pp. 277–293 (2009). https://doi.org/10.1007/978-3-642-02959-2_22
9. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Proc. IJCAR. pp. 241–255 (2012). https://doi.org/10.1007/978-3-642-31365-3_20
10. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) Proc. RTA. pp. 41–50 (2011). https://doi.org/10.4230/LIPIcs.RTA.2011.41
11. Fuhs, C., Kop, C.: A static higher-order dependency pair framework. In: Caires, L. (ed.) Proc. ESOP. pp. 752–782 (2019). https://doi.org/10.1007/978-3-030-17184-1_27

12. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. ACM TOCL **18**(2), 14:1–14:50 (2017). https://doi.org/10.1145/3060143
13. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. IPSJ Trans. Program. **1**(2), 100–121 (2008), in Japanese
14. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. JAR **58**, 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
15. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM TOPLAS **33**(2), 7:1–7:39 (2011). https://doi.org/10.1145/1890028.1890030
16. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) Proc. LPAR. pp. 301–331 (2005). https://doi.org/10.1007/978-3-540-32275-7_21
17. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press (1989)
18. Hamza, J., Voirol, N., Kunčak, V.: System FR: formalized foundations for the Stainless verifier. PACMPL **3**(OOPSLA), 166:1–166:30 (2019). https://doi.org/10.1145/3360592
19. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in Liquid Haskell. PACMPL **4**(POPL), 24:1–24:27 (2019). https://doi.org/10.1145/3371092
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) Proc. CAV. pp. 781–786 (2012). https://doi.org/10.1007/978-3-642-31424-7_64
21. Jouannaud, J.P., Rubio, A.: The higher-order recursive path ordering. In: Longo, G. (ed.) Proc. LICS. pp. 402–411 (1999). https://doi.org/10.1109/LICS.1999.782635
22. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. TCS **121**(1–2), 279–308 (1993). https://doi.org/10.1016/0304-3975(93)90091-7
23. Kojima, M., Nishida, N.: From starvation freedom to all-path reachability problems in constrained rewriting. In: Hanus, M., Inclezan, D. (eds.) Proc. PADL. pp. 161–179 (2023). https://doi.org/10.1007/978-3-031-24841-2_11
24. Kojima, M., Nishida, N.: Reducing non-occurrence of specified runtime errors to all-path reachability problems of constrained rewriting. JLAMP **135**, 100903:1–100903:19 (2023). https://doi.org/10.1016/j.jlamp.2023.100903
25. Kop, C.: Termination of LCTRSs. In: Waldmann, J. (ed.) Proc. WST. pp. 59–63 (2013). https://doi.org/10.48550/arXiv.1601.03206
26. Kop, C.: WANDA — a higher order termination tool. In: Ariola, Z.M. (ed.) Proc. FSCD. pp. 36:1–36:19 (2020). https://doi.org/10.4230/LIPIcs.FSCD.2020.36
27. Kop, C., Nishida, N.: Term rewriting with logical constraints. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) Proc. FroCoS. pp. 343–358 (2013). https://doi.org/10.1007/978-3-642-40885-4_24
28. Kop, C., Vale, D.: hezzel/cora artifact: ESOP2024 release v4 (2024). https://doi.org/10.5281/zenodo.10560907
29. Kusakari, K.: On proving termination of term rewriting systems with higher-order variables. IPSJ Trans. Program. **42**(SIG 7), 35–45 (2001), http://id.nii.ac.jp/1001/00016864/

30. Nagao, T., Nishida, N.: Rewriting induction for constrained inequalities. SCP **155**, 76–102 (2018). https://doi.org/10.1016/j.scico.2017.10.012
31. Nipkow, T.: Higher-order critical pairs. In: Kahn, G. (ed.) Proc. LICS. pp. 342–349 (1991). https://doi.org/10.48456/tr-218
32. Nishida, N., Winkler, S.: Loop detection by logically constrained term rewriting. In: Piskac, R., Rümmer, P. (eds.) Proc. VSTTE. pp. 309–321 (2018). https://doi.org/10.1007/978-3-030-03592-1_18
33. Pareto, L.: Sized types (1998), licentiate thesis. Chalmers University of Technology
34. Reddy, U.S.: Term rewriting induction. In: Stickel, M.E. (ed.) Proc. CADE. pp. 162–177 (1990). https://doi.org/10.1007/3-540-52885-7_86
35. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. JLAP **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
36. Sakata, T., Nishida, N., Sakabe, T., Sakai, M., Kusakari, K.: Rewriting induction for constrained term rewriting systems. IPSJ Trans. Program. **2**(2), 80–96 (2009), in Japanese
37. Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., Thiemann, R.: Automated termination analysis for logic programs with cut. TPLP **10**(4–6), 365–381 (2010). https://doi.org/10.1017/S1471068410000165
38. Schöpf, J., Middeldorp, A.: Confluence criteria for logically constrained rewrite systems. In: Pientka, B., Tinelli, C. (eds.) Proc. CADE. pp. 474–490 (2023). https://doi.org/10.1007/978-3-031-38499-8_27
39. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Online Trans. **4**, 114–125 (2011). https://doi.org/10.2197/ipsjtrans.4.114
40. Tait, W.W.: Intensional interpretations of functionals of finite type I. JSL **32**(2), 198–212 (1967). https://doi.org/10.2307/2271658
41. Thiemann, R., Allais, G., Nagele, J.: On the formalization of termination techniques based on multiset orderings. In: Tiwari, A. (ed.) Proc. RTA. pp. 339–354 (2012). https://doi.org/10.4230/LIPIcs.RTA.2012.339
42. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) Proc. RTA–TLCA. pp. 466–475 (2014). https://doi.org/10.1007/978-3-319-08918-8_32

Abstract Interpretation

A Modular Soundness Theory for the Blackboard Analysis Architecture

Sven Keidel[1(✉)], Dominik Helm[1,2], Tobias Roth[1,2], and Mira Mezini[1,2,3]

[1] Technische Universität Darmstadt, Darmstadt, Germany
`keidel,helm,roth,mezini@cs.tu-darmstadt.de`

[2] National Research Center for Applied Cybersecurity (ATHENE), Darmstadt, Germany

[3] Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany

Abstract. Sound static analyses are an important ingredient for compiler optimizations and program verification tools. However, mathematically proving that a static analysis is sound is a difficult task due to two problems. First, soundness proofs relate two complicated program semantics (the static and the dynamic semantics) which are hard to reason about. Second, the more the static and dynamic semantics differ, the more work a soundness proof needs to do to bridge the impedance mismatch. These problems increase the effort and complexity of soundness proofs. Existing soundness theories address these problems by deriving both the dynamic and static semantics from the same artifact, often called *generic interpreter*. A generic interpreter provides a common structure along which a soundness proof can be composed, which avoids having to reason about the analysis as a whole. However, a generic interpreter restricts which analyses can be derived, as all derived analyses must roughly follow the program execution order.

To lift this restriction, we develop a soundness theory for the blackboard analysis architecture, which is capable of describing backward, demand-driven, and summary-based analyses. The architecture describes static analyses with small independent modules, which communicate via a central store. Soundness of a compound analysis follows from soundness of all of its modules. Furthermore, modules can be proven sound independently, even though modules depend on each other. We evaluate our theory by proving soundness of four analyses: a pointer and call-graph analysis, a reflection analysis, an immutability analysis, and a demand-driven reaching definitions analysis.

1 Introduction

Developing static analyses is a laborious and complicated task due to the complexity of modern programming languages. A significant part of the complication pertains to ensuring that static analyses are *sound*, i.e., over-approximate the runtime behavior of analyzed programs. Unfortunately, even well-established static analyses are shown to be unsound, e.g., since 2010, more than 80 soundness bugs have been found in different analyses used in the LLVM compiler [46].

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-57267-8_14.

Testing helps finding soundness bugs but cannot prove their absence, leaving the trustworthiness of these analyses in question.

Mathematical soundness proofs ensure the absence of soundness bugs. However, such proofs are difficult for two reasons: First, soundness proofs relate two program semantics: the static semantics and the dynamic semantics [12]—each in its own can individually be complex. Especially modern programming language features such as reflection [30], concurrency [29], or native code [1] are notoriously difficult to analyze and hard to reason about. Second, the style of static and dynamic semantics can differ significantly, e.g., the static semantics of Doop [7], which is described in Datalog, differs significantly from dynamic semantics described with small-step rules [6]. This impedance mismatch makes soundness proofs *monolithic*, i.e., it is difficult to determine which parts of the static semantics relate to which parts of the dynamic semantics, requiring the soundness proofs to reason about both semantics as a whole. These problems complicate soundness proofs such that only leading experts with multiple years of experience can conduct them [13, 26].

To deal with the complexity of soundness proofs, existing works modularize static and dynamic semantics [5, 14, 28]. This modularization allows to compose a soundness proof for the entire analysis from soundness lemmas of small parts of the analysis. This allows reasoning about small parts of the analysis one at a time. These existing works require that both the static and dynamic semantics are derived from the same artifact, often called a *generic interpreter*. A generic interpreter describes the operational semantics of a language, without referring to details of dynamic or static semantics, and provides a common structure along which a soundness proof can be composed. However, generic interpreters restrict what types of analyses can be derived. In particular, generic interpreters derive analyses that follow the program execution order, specifically, forward whole-program abstract interpreters. But it is unclear how other types of analyses can be derived that do not follow the program execution order, such as backward, demand-driven/lazy, or summary-based analyses.

The work presented in this paper lifts this restriction by developing a soundness theory for the *blackboard analysis architecture*. The architecture is the foundation of the *OPAL framework* [21], which has been used to develop different kinds of analyses, including backward analyses [17], on-demand/lazy analyses [19,41], and summary-based analyses [21]. In the architecture, complex static analyses are modularly composed from smaller, simpler *static modules* that handle individual language features, e.g., reflection, or program properties, e.g., immutability. These modules are decoupled—they are not allowed to call each other directly; instead, they communicate with each other by exchanging information via a central data store called *blackboard* [39] orchestrated by a fixpoint solver.

To develop a soundness theory for the blackboard analysis architecture, we define a dynamic semantics, which follows the same style as the static semantics and thus avoids the impedance mismatch problem. Specifically, the dynamic semantics is composed of dynamic modules that communicate with each other via a store. Our soundness theory is *compositional*, which means that each static module can be proven sound individually and soundness for the compound analysis follows from a meta theorem. Furthermore, we extend the theory to make soundness proofs of existing static modules *reusable* across different analyses. In particular, we prove that the soundness proof of an static module remains valid, even if (a) the compound analysis processes source code elements unknown to the module and (b) the store contains other types of analysis information unknown to the module. Furthermore, our proofs are polymorphic in the lattices on which static modules operate, i.e., the lattices can be changed without affecting soundness. For instance, we can reuse a pointer-static module, which typically depends on an allocation-site lattice, in a reflection analysis to propagate string information by extending this lattice without invalidating the pointer-static modules' soundness proof.

We demonstrate the applicability of our theory by implementing four different analyses and their dynamic semantics in the blackboard analysis architecture: (1) a pointer and call-graph analysis, (2) an analysis for reflection, (3) an immutability analysis, and (4) a demand-driven reaching-definitions analysis. Our choice of analyses is inspired by existing state-of-the-art analyses for Java implemented in the OPAL framework [21, 41]. We implemented and tested each analysis and dynamic semantics in Scala to ensure they are executable. Furthermore, we used our theory to prove each analysis sound, where each analysis exercises a different aspect of our theory: (1) static modules can be proven sound independently despite mutually depending on each other, (2) soundness of modules remains valid even though the lattice changes, (3) soundness of a module remains valid even though different source code elements are analyzed, and (4) our theory applies to analyses which do not follow the program execution order.

In summary, we make the following contributions:

- We give the first formalization of the blackboard analysis architecture (Section 2).
- We develop a theory of compositional soundness proofs for the formal model of the blackboard analysis architecture. We prove that soundness of an analysis follows from independent soundness proofs for each of its modules (Section 3).
- We show how to make soundness proofs reusable by extending our theory (Section 4).
- We demonstrate the applicability of our theory on four different types of analyses (Section 5).

All proofs of theorems, lemmas, and case studies are provided in the paper's supplementary material.

2 Blackboard Analysis Architecture

In this section, we introduce and formalize the static and dynamic semantics of the blackboard analysis architecture used in the OPAL framework [21].

2.1 Static Semantics

Static analyses in the blackboard analysis architecture consist of multiple *static modules* exchanging information via a central data store called *blackboard* [39]. This avoids coupling between modules as they are not allowed to call each other directly: Modules store analysis results in the blackboard using keys. These keys allow other modules to retrieve results without needing to know their producer.

Definition 1 (Static Semantics). *We define basic notions and datatypes of the static semantics of the blackboard analysis architecture:*

1. *Entities ($\widehat{e} \in \widehat{\mathsf{Entity}}$)*[4] *are parts of programs an analysis can compute information for. For example, entities could be classes, methods, statements, fields, variables, or allocation sites of objects. Entities are ordered discretely: $\widehat{e}_1 \sqsubseteq \widehat{e}_2$ iff $\widehat{e}_1 = \widehat{e}_2$.*
2. *Kinds ($\kappa \in \mathsf{Kind}$) identify analysis information that can be computed for an entity. For example, a class entity could have kinds for its immutability or thread safety, a variable entity could have kinds for its definition site or approximations of its value. Kinds are also ordered discretely.*
3. *Properties ($\widehat{p} \in \widehat{\mathsf{Property}}[\kappa]$ where $\widehat{\mathsf{Property}} : \mathsf{Kind} \to \mathsf{Lattice}$) denote analysis information which is identified by a kind κ. For instance, a class entity could have an immutability property "$\mathsf{mutable}$" or "$\mathsf{immutable}$". Properties of a kind are partially ordered and form a lattice.*
4. *A central store ($\widehat{\sigma} \in \widehat{\mathsf{Store}} \subseteq \widehat{\mathsf{Entity}} \times (\kappa : \mathsf{Kind}) \rightharpoonup \widehat{\mathsf{Property}}[\kappa]$)*[5] *contains all properties for each entity and kind. We use the notation $\widehat{\sigma}(\widehat{e}, \kappa)$ for a store lookup of an entity $\widehat{e}$ and kind κ, which results in the bottom element $\bot$ in case the property is not present. Furthermore, we use the notation $\widehat{\sigma} \sqcup [\widehat{e}, \kappa \mapsto \widehat{p}]$ for writing a new property $\widehat{p}$ to the store. If a property for the entity $\widehat{e}$ and κ already exists in the store, then the old property is joined with the new property. Stores are ordered point-wise.*
5. *Static modules ($\widehat{f} \in \widehat{\mathsf{Module}} = \widehat{\mathsf{Entity}} \times \widehat{\mathsf{Store}} \to \widehat{\mathsf{Store}}$) are monotone functions that compute properties of a given entity. The store allows multiple static modules to communicate and exchange information without having to call each other directly. Each static module has access to the entire store and can contribute to one or more properties.*
6. *The fixpoint algorithm ($\mathsf{fix} : \mathcal{P}(\widehat{\mathsf{Module}}) \times \widehat{\mathsf{Store}} \to \widehat{\mathsf{Store}}$) computes a fixpoint of a compound analysis $\widehat{F} \in \mathcal{P}(\widehat{\mathsf{Module}})$ for an initial store $\widehat{\sigma}_1$. More specifically, the fixpoint $\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1)$ is a store $\widehat{\sigma}_n \sqsupseteq \widehat{\sigma}_1$ such that static modules $\widehat{f} \in \widehat{F}$ do not add new information, i.e., $\widehat{f}(\widehat{e}, \widehat{\sigma}_n) = \widehat{\sigma}_n$ for all $\widehat{e} \in dom(\widehat{\sigma}_n)$. The fixpoint is unique and guaranteed to exist when all properties are lattices of finite height [10].*

[4] We use a hat symbol $\widehat{\ }$ to disambiguate static definitions from dynamic definitions with the same name but without hat.

[5] The syntax $A \rightharpoonup B$ denotes a partial function from A to B. Furthermore, $dom(f)$ is the set of all inputs for which a partial function f is defined.

The types $\widehat{\mathsf{Entity}}$*,* Kind*, and* $\widehat{\mathsf{Property}}$ *are defined by analysis developers, whereas the other types and functions are fixed by this definition.* □

We illustrate Definition 1 at the example of a text-book reaching-definitions analysis [38] for an imperative language with labeled assignments and expressions:

$\widehat{\mathsf{Entity}} = \mathsf{Stmt}$
$\widehat{\mathsf{Property}}[\kappa_{\mathtt{ControlFlowPred}}] = \mathcal{P}(\mathsf{Stmt})$
$\widehat{\mathsf{Property}}[\kappa_{\mathtt{ReachingDefs}}] = \mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign})$
$\widehat{\mathsf{Store}} = [\mathsf{Stmt} \times \kappa_{\mathtt{ControlFlowPred}} \rightharpoonup \mathcal{P}(\mathsf{Stmt})]$
$\quad \cup [\mathsf{Stmt} \times \kappa_{\mathtt{ReachingDefs}} \rightharpoonup (\mathsf{Var} \rightharpoonup \mathcal{P}(\mathsf{Assign}))]$

```
$\widehat{\mathsf{reachingDefs}}$(stmt: $\widehat{\mathsf{Entity}}$, $\widehat{\sigma}$: $\widehat{\mathsf{Store}}$): $\widehat{\mathsf{Store}}$ =
  predecessors = $\widehat{\sigma}$(stmt, $\kappa_{\mathtt{ControlFlowPred}}$)
  $\widehat{\mathtt{in}}$ = $\bigsqcup_{\mathtt{p} \in \mathtt{predecessors}} \widehat{\sigma}$(p, $\kappa_{\mathtt{ReachingDefs}}$)
  $\widehat{\mathtt{out}}$ = stmt match
    case Assign(x,_,_) => $\widehat{\mathtt{in}}[\mathtt{x} \mapsto \{\mathtt{stmt}\}]$
    case _ => $\widehat{\mathtt{in}}$
  $\widehat{\sigma} \sqcup [\mathtt{stmt}, \kappa_{\mathtt{ReachingDefs}} \mapsto \widehat{\mathtt{out}}]$
```

The static module $\widehat{\mathsf{reachingDefs}}$ is implemented with Scala-like pseudo code. Module $\widehat{\mathsf{reachingDefs}}$ computes for every statement of the program which variable definitions reach it. Therefore, entities are statements and the module's property is a mapping from variables to assignments that may have defined it. Module $\widehat{\mathsf{reachingDefs}}$ joins the reaching definitions of all control-flow predecessors and then updates them on variable assignments. Note that module $\widehat{\mathsf{reachingDefs}}$ neither computes the control-flow predecessors directly nor does it call another module which computes this information. Instead, it retrieves this information from the store $\widehat{\sigma}$. This decoupling avoids dependencies between static modules and enables compositional soundness proofs.

2.2 Dynamic Semantics

Static analyses in the blackboard analysis architecture are proven sound with respect to a dynamic semantics in the same style, which we define formally in this subsection:

Definition 2 (Dynamic Semantics). *We define the dynamic semantics used to prove soundness of analyses in the blackboard analysis architecture:*

1. *The dynamic semantics depends on concrete versions of* entities *(*$e \in \mathsf{Entity}$*),* properties *(*$p \in \mathsf{Property}[\kappa]$ *where* $\mathsf{Property} : \mathsf{Kind} \to \mathsf{Set}$*) and* stores *(*$\sigma \in \mathsf{Store} \subseteq \mathsf{Entity} \times (\kappa : \mathsf{Kind}) \to \mathsf{Property}[\kappa]$*). The kinds are the same as for static modules.*
2. *Dynamic modules (*$f \in \mathsf{Module} = \mathsf{Entity} \times \mathsf{Store} \rightharpoonup \mathsf{Store}$*) are partial functions which may only be defined for a subset of entities. Furthermore, the partial function is undefined in case it tries to lookup an element from the store which is not present.*

3. *Static analyses are proven sound with respect to a dynamic reachability semantics (*$\mathsf{reachable} : \mathcal{P}(\mathsf{Module}) \times \mathsf{Store} \to \mathcal{P}(\mathsf{Store})$*). The reachability semantics returns the set of all reachable stores by iteratively applying a set of dynamic modules. More specifically, the set* $\mathsf{reachable}(F, \sigma_1)$ *contains store* σ_1 *and for all* $f \in F$*, reachable stores* σ*, and for entities* $e \in dom(\sigma)$*, the set contains* $f(e, \sigma)$*, if it is defined.* $\square$

We illustrate these definitions again at the example of the reaching-definitions analysis which we introduced in the previous subsection:

$\mathsf{Entity} = \mathsf{Stmt} \mid \mathsf{Unit}$
$\mathsf{Property}[\kappa_{\mathtt{ControlFlowPred}}] = \mathsf{Stmt}$
$\mathsf{Property}[\kappa_{\mathtt{ReachingDefs}}] = \mathsf{Var} \rightharpoonup \mathsf{Assign}$
$\mathsf{Property}[\kappa_{\mathtt{State}}] = \mathtt{ProgramState}$
$\mathsf{Store} = [\mathsf{Stmt} \times \kappa_{\mathtt{ControlFlowPred}} \rightharpoonup \mathsf{Stmt}] \cup [\mathsf{Stmt} \times \kappa_{\mathtt{ReachingDefs}} \rightharpoonup (\mathsf{Var} \rightharpoonup \mathsf{Assign})]$
$\quad \cup [\mathsf{Unit} \times \kappa_{\mathtt{State}} \rightharpoonup \mathsf{ProgramState}]$

```
reachingDefs(stmt: Entity, $\sigma$: Store): Store =
  predecessor = $\sigma$(stmt, $\kappa_{\mathtt{ControlFlowPred}}$)
  in = $\sigma$(predecessor, $\kappa_{\mathtt{ReachingDefs}}$)
  out = stmt match
    case Assign(x,_,_) => in[x $\mapsto$ stmt]
    case _ => in
  $\sigma$[stmt, $\kappa_{\mathtt{ReachingDefs}}$ $\mapsto$ out]

controlFlow(stmt1: Entity, $\sigma$: Store): Store =
  state1 = $\sigma$[Unit, $\kappa_{\mathtt{State}}$]
  (stmt2, state2) = step(stmt1, state1)
  $\sigma$[stmt2, $\kappa_{\mathtt{ControlFlowPred}}$ $\mapsto$ stmt1][Unit, $\kappa_{\mathtt{State}}$ $\mapsto$ state2]
```

Dynamic module reachingDefs is analogous to its static counterpart $\widehat{\mathsf{reachingDefs}}$, but computes the *most recent* definition of a variable instead of all possible definitions. The dynamic module depends on the control-flow predecessor, which is the most recently executed statement. The control-flow predecessors are computed by module controlFlow, which is based on a small-step operational semantics $\mathtt{step} : \mathtt{Stmt} \times \mathtt{ProgramState} \rightharpoonup \mathtt{Stmt} \times \mathtt{ProgramState}$. Module controlFlow demonstrates that the blackboard architecture is capable to integrate existing dynamic operational semantics, such as those for Java [6] or WebAssembly [18].

The blackboard analysis architecture not only modularizes the static semantics but also the dynamic semantics, which is crucial for enabling compositional and reusable soundness proofs. In particular, each static module is proven sound with respect to exactly one dynamic module, which limits the proof scope and guarantees proof independence. Furthermore, for analyses that approximate non-standard dynamic semantics, the standard dynamic semantics can be modularly extended with further modules (e.g., Section 5.1).

To summarize, in this section we formally defined the blackboard analysis architecture, which allows to implement static analyses modularly. Furthermore, we defined a dynamic semantics in the same style against which analyses are proven sound.

3 Compositional Soundness Proofs

In this section, we develop a theory of compositional soundness proofs for analyses in the blackboard style: Soundness of a compound analysis follows directly from soundness of the individual static modules. This soundness theory simplifies the soundness proof, because it allows analysis developers to focus on soundness of individual static modules, instead of having to reason about soundness of the interaction of all static modules with each other. Furthermore, the soundness theory makes the proofs more maintainable, as a change to a module only affects the proof of that module and nothing else.

We start the section by defining soundness of static modules and then work up to soundness of whole analyses. The definitions of soundness are standard and build upon the theory of *abstract interpretation* [12]:

Definition 3 (Soundness of Static Modules). *An static module $\widehat{f} \in \widehat{\mathsf{Module}}$ is sound if it overapproximates its dynamic counterpart $f \in \mathsf{Module}$:*

$$\mathsf{sound}(f, \widehat{f}) \;\; \textit{iff} \; \forall \widehat{e} \in \widehat{\mathsf{Entity}}, \widehat{\sigma} \in \widehat{\mathsf{Store}}, e \in \gamma_{\mathsf{Entity}}(\widehat{e}), \sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma}). \\ f(e, \sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e}, \widehat{\sigma})) \qquad \square$$

The expression $x \in \gamma(\widehat{y})$ reads as "element $\widehat{y}$ soundly overapproximates the concrete element x." Function $\gamma : \widehat{L} \to \mathcal{P}(L)$ is a monotone function from an abstract domain $\widehat{L}$ to a powerset of a concrete domain L and is called *concretization function*. We do not require that an *abstraction function* $\alpha : \mathcal{P}(L) \to \widehat{L}$ in the opposite direction exists nor that γ and α form a Galois connection, both of which are not necessary for soundness proofs.

The soundness definition above requires that analysis developers define concretizations for entities ($\gamma_{\mathsf{Entity}} : \widehat{\mathsf{Entity}} \to \mathcal{P}(\mathsf{Entity})$) and properties ($\gamma_{\mathsf{Property}} : \widehat{\mathsf{Property}}[\kappa] \to \mathcal{P}(\mathsf{Property}[\kappa])$). Often the abstract and concrete entities are of the same type ($\widehat{\mathsf{Entity}} = \mathsf{Entity}$). In this case, the concretization functions map to singleton sets ($\gamma_{\mathsf{Entity}}(e) = \{e\}$). Based on concretization functions for entities, kinds, and properties, we define a point-wise concretization on stores. The definition can be found in the supplementary material.

In the following, we define soundness of compound analyses.

Definition 4 (Soundness of a Compound Analysis). *Let $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ be a set of static modules paired with corresponding dynamic modules. A compound analysis is sound if the fixpoint of all of its static modules overapproximates the reachability semantics of the corresponding dynamic modules:*

$$\mathsf{sound}(\Phi) \;\; \textit{iff} \;\; \forall \widehat{\sigma} \in \widehat{\mathsf{Store}}.\;\; \mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma})) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma})) \\ \text{where } F = \{f \mid (f, \_) \in \Phi\} \text{ and } \widehat{F} = \{\widehat{f} \mid (\_, \widehat{f}) \in \Phi\}. \qquad \square$$

The compound analysis approximates the dynamic reachability semantics (Definition 2.3), which collects the set of all stores reachable by applying dynamic modules. The dynamic reachability semantics is a collecting semantics, commonly used to prove soundness of abstract interpreters [12].

We are now ready to state the main theorem of this work:

Theorem 1 (Soundness Composition). *Let $\Phi \subseteq \mathsf{Module} \times \widehat{\mathsf{Module}}$ be a set of static modules paired with corresponding dynamic modules. Soundness of a compound analysis follows from soundness of all of its static modules:*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ for all } (f, \widehat{f}) \in \Phi \text{ then } \mathsf{sound}(\Phi).$$

Proof. *We show* $\mathsf{reachable}(F, \gamma_{\mathsf{Store}}(\widehat{\sigma}_1)) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *by well-founded induction on* $X \preceq \mathsf{reachable}(F, X)$.

- *Base case:* $\mathsf{reachable}(F, \varnothing) = \varnothing \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$
- *Inductive case: Suppose* $X \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *and* $\widehat{\sigma}_n = \mathsf{fix}(\widehat{F}, \widehat{\sigma}_1)$. *Then for all* $\sigma \in X \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$, *we get* $dom(\sigma) \subseteq \gamma_{\mathsf{Entity} \times \mathsf{Kind}}(dom(\widehat{\sigma}_n))$ *and* $\sigma(e, k) \in \gamma_{\mathsf{Property}}(\widehat{\sigma}_n(\widehat{e}, \kappa))$ *for all* $\forall(\widehat{e}, \kappa) \in dom(\widehat{\sigma}_n)$ *and* $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. *Furthermore, since* $\widehat{\sigma}_n$ *is a fixpoint, it holds* $\widehat{f}(\widehat{e}, \widehat{\sigma}_n) \sqsubseteq \widehat{\sigma}_n$ *for all* $\widehat{f} \in \widehat{F}$ *and* $\widehat{e} \in dom(\widehat{\sigma}_n)$. *From* $\mathsf{sound}(f, \widehat{f})$ *we conclude* $f(e, \sigma) \in \gamma_{\mathsf{Store}}(\widehat{f}(\widehat{e}, \widehat{\sigma}_n)) \subseteq \gamma_{\mathsf{Store}}(\widehat{\sigma}_n) = \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$ *for all* $(f, \widehat{f}) \in \Phi$, $(e, \_) \in dom(\sigma)$, $(\widehat{e}, \_) \in dom(\widehat{\sigma}_n)$ *with* $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. *It follows* $\mathsf{reachable}(F, X) \subseteq \gamma_{\mathsf{Store}}(\mathsf{fix}(\widehat{F}, \widehat{\sigma}_1))$.

□

We illustrate this theorem by applying it to the reaching definitions analysis from Section 2.1. Specifically, soundness of the compound analysis follows from soundness of module $\widehat{\mathsf{reachingDefs}}$ module $\widehat{\mathsf{controlFlow}}$ by Theorem 1:

$$\frac{\mathsf{sound}(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}}) \qquad \mathsf{sound}(\mathsf{controlFlow}, \widehat{\mathsf{controlFlow}})}{\mathsf{sound}(\{(\mathsf{reachingDefs}, \widehat{\mathsf{reachingDefs}}), (\mathsf{controlFlow}, \widehat{\mathsf{controlFlow}})\})}$$

This means $\widehat{\mathsf{reachingDefs}}$ can be proven sound independently from $\widehat{\mathsf{controlFlow}}$, even though the modules interact with each other in the compound analysis. The proof independence is possible because neither module $\mathsf{reachingDefs}$ nor $\widehat{\mathsf{reachingDefs}}$ call the control-flow modules directly. Instead, both the static and dynamic module read the control-flow information from the stores, which are guaranteed to be a sound overapproximation initially (assumption $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$). Furthermore, only properties that the reaching-definitions modules themselves wrote to the store need to be sound overapproximations. Properties that other modules wrote to the store are not subject of the soundness proof of the reaching-definitions modules. The soundness proof of module $\widehat{\mathsf{reachingDefs}}$ is found in the supplementary material.

To summarize, in this section we developed a theory of compositional soundness proofs for analyses described in the blackboard architectural style. Each static module can be proven sound independently from other modules. Furthermore, soundness of a whole analysis follows directly from soundness of each module. In particular, no reasoning about the analysis as a whole is required.

4 Reusable Soundness Proofs

As of now, static modules refer to a specific type of entities, kinds, properties, and stores. However, adding new modules to an analysis may require extending these types. This invalidates the soundness proofs of existing modules and they need to be re-established. In this section, we extend our theory to make static modules and their soundness proofs reusable.

4.1 Extending the Type of Entities and Kinds

We start by explaining how entities and kinds can be extended without invalidating existing soundness proofs.

For example, if we were to add a taint static module to an existing analysis over types $\widehat{\mathsf{Entity}}$, Kind, and $\widehat{\mathsf{Store}}$, we needed to extend these types to hold the new analysis information:

$$\widehat{\mathsf{Entity}}' = \widehat{\mathsf{Entity}} \mid \mathsf{Var} \qquad\qquad \mathsf{Kind}' = \mathsf{Kind} \mid \kappa_{\mathtt{Taint}}$$

But this invalidates the proofs of existing modules that depend on the subsets $\widehat{\mathsf{Entity}}$ and Kind. To solve this problem, we first parameterize the type of modules to make explicit what types of entities and kinds they depend on:

Definition 5 (Parameterized Modules (Preliminary)). *We define a type of module that is parameterized by the types of entities E, kinds K, and store S:*

$$f \in \mathsf{Module}[E, K] = \forall S : \mathsf{Store}[E, K].\; E \times S \to S \qquad \square$$

Interface $\mathsf{Store}[E, K]$ defines read and write operations for an abstract store type S, that restricts access to entities of type E and kinds of type K. The store interface allows us to call parameterized modules with stores containing supersets of the type of entities and kinds.

For these parameterized modules, we define a sound *lifting* to supersets of entities and kinds:

```
lift : $\forall E', K', E \subseteq E', K \subseteq K', \mathsf{Module}[E, K] \to \mathsf{Module}[E', K']$
lift($f$)($e'$, $\sigma$) = $e'$ match
  case $e : E$ => $f$($e$, $\sigma$)
  case _ => $\sigma$
```

The lifting calls module f on all entities of type E on which f is defined and simply ignores all other entities, returning the store unchanged. For example, the lifted reaching-definitions module $\mathsf{lift}[\mathsf{Stmt} \mid \mathsf{Var},\ \kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}} \mid \kappa_{\mathtt{Taint}}](\widehat{\mathsf{reachingDefs}})$ operates on the entities Stmt and the kinds $\kappa_{\mathtt{ReachingDefs}} \mid \kappa_{\mathtt{ControlFlowPred}}$, but ignores entities Var and kinds $\kappa_{\mathtt{Taint}}$.

The lifting preserves soundness of the lifted modules for disjoint extensions of entities.

Definition 6 (Disjoint Extension). *Entities $\widehat{E}' \supseteq \widehat{E}$ and $E' \supseteq E$ are a disjoint extension iff $\gamma_{\mathsf{Entity}}(\widehat{E}) \subseteq E$ and $\gamma_{\mathsf{Entity}}(\widehat{E}' \setminus \widehat{E}) \subseteq E' \setminus E$.* $\square$

In other words, the concretization function γ_{Entity} does not mix up entities in $\widehat{E}$ and $\widehat{E}' \setminus \widehat{E}$.

Lemma 1 (Lifting preserves Soundness). *Let $\widehat{f} \in \mathsf{Module}[\widehat{E}, K]$ and $f \in \mathsf{Module}[E, K]$ be a parameterized static module and dynamic module, $\widehat{E}' \supseteq \widehat{E}$ and $E' \supseteq E$ be a disjoint extension of entities, and $K' \supseteq K$ a superset of kinds.*

$$\text{If } \mathsf{sound}(f, \widehat{f}) \text{ then } \mathsf{sound}(\mathsf{lift}[E', K'](f), \mathsf{lift}[\widehat{E}', K'](\widehat{f})).$$

Proof. Let $\widehat{f}$: $\mathsf{Module}[\widehat{E}, K]$ and f : $\mathsf{Module}[E, K]$ be an analysis and dynamic module. Furthermore, let $\widehat{e}$: $\widehat{E}'$ and $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$ be an entity and $\widehat{\sigma}$: $\mathsf{Store}[\widehat{E}', K']$ and $\sigma \in \gamma_{\mathsf{Store}}(\widehat{\sigma})$ be an abstract and concrete store.

- *In case $\widehat{e} \in \widehat{E}$ then also $e \in E$. Hence, $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(e, \sigma)$. Soundness follows by $\mathsf{sound}(f, \widehat{f})$.*
- *In case $\widehat{e} \in \widehat{E}' \setminus \widehat{E}$ then also $e \in E' \setminus E$ for all $e \in \gamma_{\mathsf{Entity}}(\widehat{e})$. Hence $\mathsf{lift}(\widehat{f})(\widehat{e}, \widehat{\sigma}) = \widehat{f}(\widehat{e}, \widehat{\sigma})$ and $\mathsf{lift}(f)(e, \sigma) = f(\widehat{e}, \widehat{\sigma})$.* □

This lemma means that we can prove the soundness of static modules once for specific types of entities and kinds. Later, we can reuse the modules in a compound analysis with extended entities and kinds without having to prove soundness again.

4.2 Changing the Type of Properties

Next, we extend our theory to allow changing the type of properties without invalidating the soundness proofs of existing modules that use them.

For example, consider we already have a pointer-static module that propagates object allocation information $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}}$. We may want to track string information as well. This could be done with a independent string-tracking static module with its own lattice. However, since tracking strings is mostly identical to tracking pointer information, such an additional module would duplicate significant amounts of code and require a new proof from scratch.

Instead, we can thus reuse the same pointer-static module to propagate string information $\widehat{\mathsf{Str}}$ by changing its lattice to $\widehat{\mathsf{Property}}'[\kappa_{\mathtt{Val}}] = \widehat{\mathsf{Obj}} \times \widehat{\mathsf{Str}}$. However, this invalidates the soundness proof of the pointer-static module as it depends on type $\widehat{\mathsf{Property}}[\kappa_{\mathtt{Val}}]$.

To solve this problem, we generalize the type of static modules again to be polymorphic over the type $\widehat{\mathsf{Property}}$:

Definition 7 (Parameterized Modules (Final)). *We define a type of module that is parameterized by the type of entities E, kinds K, properties P, and stores S:*

$$f \in \mathsf{Module}[E, K, I] = \forall P : I, S : \mathsf{Store}[E, K, P], E \times S \to S \qquad \square$$

Interface $\mathsf{Store}[E, K, P]$ restricts access to entities of type E and type K and contains properties of type P. Interface I defines operations on properties P.

For example, a pointer-static module may depend on the Scala-like interface $\mathsf{Objects}$ in Listing 1.1. Interface $\mathsf{Objects}$ depends on a type variable `Value`, which refers to possible values of variables. Function newObj creates a new object of a

```
trait Objects[Value]:
  newObj(class: Class, ctx: Context): Value
  forObj[S](Value, S)(f: (Class, Context, S) => S): S

object $\widehat{\text{AllocationSite}}$ extends Objects[$\widehat{\text{Obj}}$]:
  newObj(class, ctx) = {(class, ctx)}
  forObj[S]($\widehat{\text{Obj}}$(objs), $\widehat{\sigma}$)(f) = $\bigsqcup_{\text{(class,ctx)}\in\text{objs}}$ f(class,ctx,$\widehat{\sigma}$)

object $\widehat{\text{AllocationSiteAndStrings}}$ extends Objects[$\widehat{\text{Obj}} \times \widehat{\text{Str}}$]:
  newObj(class, ctx) = ({(class, ctx)}, $\bot$)
  forObj[S](value, $\widehat{\sigma}$)(f) = value match
    case (objs,_) => $\bigsqcup_{\text{(class,ctx)}\in\text{objs}}$ f(class, ctx, $\widehat{\sigma}$)
```

Listing 1.1: Interface for different Object Abstractions

certain class and context. Function forObj iterates over all such objects applying continuation f. Continuation f takes a class name, context, and store and returns a modified store. Interface Objects can be instantiated to support different value abstractions. For example, instance $\widehat{\text{AllocationSite}}$ implements the interface with an allocation-site abstraction $\widehat{\text{Obj}} = \widehat{\text{Obj}}(\mathcal{P}(\text{Class} \times \text{Context}))$ which abstracts object allocations by their class names and a call string to their allocation site. Instance $\widehat{\text{AllocationSiteAndStrings}}$ implements a reduced product [9] of objects $\widehat{\text{Obj}}$ and strings $\widehat{\text{Str}} = \texttt{Constant}[\texttt{String}]$, which abstracts the value of strings with a constant abstraction. This allows us to reuse the same pointer-static module to propagate string information.

Note that certain interfaces may restrict what instances can be implemented. For example, an abstract domain that only approximates strings but not objects, cannot soundly implement operation forObj in interface Objects. In this case, interfaces need to be generalized to allow a wider range of instances.

4.3 Soundness of Parameterized Modules

In this subsection, we define soundness of parameterized static modules and prove a generalized soundness composition theorem.

Definition 8 (Soundness of Parameterized static Modules). *A parameterized static module $\widehat{f} : \widehat{\text{Module}}[\widehat{E}, K, I]$ is sound w.r.t. a parameterized dynamic module $f : \text{Module}[E, K, I]$ iff all their instances are sound:*

$$\text{sound}(f, \widehat{f}) \textit{ iff } \forall P : I, \widehat{P} : I, S : \text{Store}[E, K, P], \widehat{S} : \text{Store}[\widehat{E}, K, \widehat{P}].$$
$$\text{sound}(P, \widehat{P}) \implies \text{sound}(f[P, S], \widehat{f}[\widehat{P}, \widehat{S}]). \qquad \Box$$

Parameterized static modules are proven sound for all sound instances of property interface I. A static instance $\widehat{P} : I$ is sound w.r.t. to a dynamic instance $P : I$, if all of its operations are sound. Soundness for dynamic and static instances of interface $\mathsf{Objects}$ in Listing 1.1 is defined as follows:

$$\begin{aligned}&\mathsf{sound}(\mathsf{newObj}, \widehat{\mathsf{newObj}}) \text{ iff } \forall c, \widehat{h}, h \in \gamma(\widehat{h}), \mathsf{newObj}(c, h) \in \gamma_{\mathsf{Obj}}(\widehat{\mathsf{newObj}}(c, \widehat{h}))\\ &\mathsf{sound}(\mathsf{forObj}, \widehat{\mathsf{forObj}}) \text{ iff } \forall f, \widehat{f}, \mathsf{sound}(f, \widehat{f}) \implies \mathsf{sound}(\mathsf{forObj}(f), \widehat{\mathsf{forObj}}(\widehat{f}))\end{aligned}$$

Soundness of first-order operations like $\widehat{\mathsf{newObj}}$ is similar to that of static modules (Definition 3). Soundness of higher-order operations like $\widehat{\mathsf{forObj}}$ is proven w.r.t. all sound functions $\widehat{f}$.

Finally, we generalize the soundness composition Theorem 1 to parameterized static modules. In particular, an analysis composed of parameterized static modules is sound if all of its modules are sound and the instance of its property interface is sound.

Theorem 2 (Soundness Composition for Parameterized Static Modules). *Let Φ be parameterized static modules paired with corresponding dynamic modules over families of entities $\widehat{E}' = \bigcup_i \widehat{E}_i, E' = \bigcup_i E_i$, kinds $K' = \bigcup_i K_i$, properties $\widehat{P}$, P.*

$$\begin{aligned}&\text{If } \mathsf{sound}(f, \widehat{f}) \text{ for all } (f, \widehat{f}) \in \Phi \text{ and } \mathsf{sound}(P, \widehat{P}) \text{ then } \mathsf{sound}(\Phi'),\\ &\text{where } \Phi' = \{(\mathsf{lift}[E', K'](f), \mathsf{lift}[\widehat{E}', K'](\widehat{f})) \mid (f, \widehat{f}) \in \Phi\}\end{aligned}$$

Proof. We instantiate the polymorphic modules $f, \widehat{f}$ with the compound types to obtain $\mathsf{sound}[E', K'](\mathsf{lift}(f), \mathsf{lift}[E', K'](\widehat{f}))$. Then the soundness composition Theorem 3.4 for monomorphic modules applies. □

To summarize, in this section we explained how the type of entities, kinds, and properties can be changed without invalidating the soundness proofs of existing modules. To this end, we generalized the type of modules to be parametric over the type of entities, kinds, and properties. The parameterized modules access properties via an interface. The instances of this interface are specific to certain types of properties and require a soundness proof.

5 Applicability of the Theory

In this section, we demonstrate the applicability of our theory by first developing four analyses in the blackboard architecture and then proving them sound compositionally.

5.1 Case Studies

We developed four different analyses in the blackboard architecture (Section 2) together with their dynamic semantics (Section 2.2). We proved each analysis sound and discuss the proofs in Section 5.2. Each analysis exercises a specific part of our soundness theory:

– A pointer analysis which mutually depends on a call-graph analysis (exercises the part of our theory presented in Section 3).

- A reflection analysis which reuses the pointer analysis to propagate string information (exercises Section 4.2).
- A field and object immutability analysis depending on all above analyses (exercises Section 4.1).
- A demand-driven reaching-definitions analysis which demonstrates that our theory applies to this type of analyses.

Our choice of analyses was inspired by similar but more complex analyses for JVM-bytecode implemented in OPAL, which scale to real-world applications [21, 41]. Our analyses operate on a simpler object-oriented language with the following abstract syntax:

$$\begin{aligned}
&\mathsf{Class} = \mathsf{Class}(\mathsf{ClassName}, \mathsf{ClassName}, \mathsf{Field}^*, \mathsf{Method}^*)\\
&\mathsf{Method} = \mathsf{SourceMethod}(\mathsf{MethodName}, \mathsf{Var}^*, \mathsf{Stmt}^*) \mid \mathsf{NativeMethod}(\mathsf{MethodName})\\
&\mathsf{Stmt} = \mathsf{Assign}(\mathsf{Ref}, \mathsf{Expr}) \mid \mathsf{Return}(\mathsf{Method}, \mathsf{Expr}) \mid \mathsf{If}(\mathsf{Expr}, \mathsf{Stmt}^*, \mathsf{Stmt}^*)\\
&\qquad \mid \mathsf{While}(\mathsf{Expr}, \mathsf{Stmt}^*)\\
&\mathsf{Expr} = \mathsf{Ref} \mid \mathsf{New}(\mathsf{ClassName}, (\mathsf{Field} \times \mathsf{Expr})^*) \mid \mathsf{StringLit}(\mathsf{String}) \mid \mathsf{Concat}(\mathsf{Expr}, \mathsf{Expr})\\
&\qquad \mid \mathsf{Call}(\mathsf{Expr}, \mathsf{MethodName}, \mathsf{Expr}^*) \mid \mathsf{BoolLit}(\mathsf{Bool}) \mid \mathsf{Equals}(\mathsf{Expr}, \mathsf{Expr})\\
&\mathsf{Ref} = \mathsf{VarRef}(\mathsf{Var}) \mid \mathsf{FieldRef}(\mathsf{Ref}, \mathsf{Field})
\end{aligned}$$

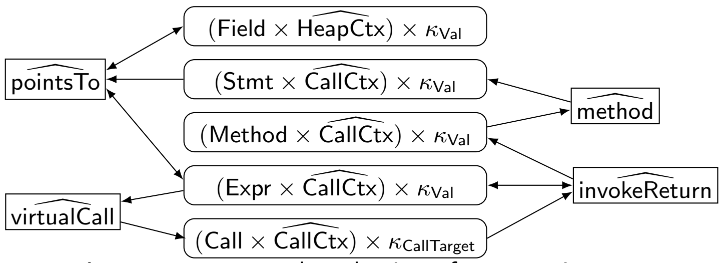
The language features inheritance, mutable memory, class fields, virtual method calls, and Java-like reflection [35]. Reflection is modeled as virtual calls to native methods. We also deliberately added features such as control-flow constructs and boolean operations. These are ignored by the analyses, but need to be modeled by dynamic semantics, complicating the soundness proof of the analyses.

We implemented and tested each analysis in Scala to ensure they are executable. Furthermore, we implemented and tested the corresponding dynamic semantics to ensure they are sensible. The code of analyses and dynamic semantics can be found in the supplementary material accompanying this paper. In the following, we discuss the implementation of each analysis in more detail.

Pointer and Call-Graph Analysis A pointer analysis for an object-oriented language computes which objects a variable or field may point to. A call-graph analysis determines which methods may be called at specific call sites. Pointer and call-graph analyses are the foundation which many other analyses build upon.

The analyses are composed from four static modules, whose dependencies are visualized in Figure 1. An arrow from a store entry to a module represents a read, an arrow in the other direction represents a write. Even though all modules implicitly depend on each other, they can be proven sound independently from each other (Section 3). This is possible because they do not call other modules directly, instead, all communication happens via the store.

Module $\widehat{\mathsf{method}}$ registers each statement of a method in the store to trigger other modules. It disregards control flow as the analysis is flow-insensitive and hence also registers statements that can never be executed. Flow-insensitive


(Field × $\widehat{\mathsf{HeapCtx}}$) × κ_{Val}
$\widehat{\mathsf{pointsTo}}$
(Stmt × $\widehat{\mathsf{CallCtx}}$) × κ_{Val}
$\widehat{\mathsf{method}}$
(Method × $\widehat{\mathsf{CallCtx}}$) × κ_{Val}
(Expr × $\widehat{\mathsf{CallCtx}}$) × κ_{Val}
$\widehat{\mathsf{invokeReturn}}$
$\widehat{\mathsf{virtualCall}}$
(Call × $\widehat{\mathsf{CallCtx}}$) × $\kappa_{\mathsf{CallTarget}}$

Arrows represent reads and writes of store entries

Fig. 1: Points-To and Call-Graph Static Modules

analyses can be more performant than flow-sensitive ones, but traditional approaches using generic abstract interpreters do not allow for flow-insentitive analyses. Module $\widehat{\mathsf{pointsTo}}$ analyzes `New` expressions and assignments of variable and field references. Module $\widehat{\mathsf{virtualCall}}$ resolves target methods of virtual calls based on the receiver object. Once a call is resolved, module $\widehat{\mathsf{invokeReturn}}$ extends the call context, assigns the method parameters and return value. Finally, it registers the called method as an entity in the store, triggering module $\widehat{\mathsf{method}}$.

The entities of the analyses are fields, statements, expressions, methods, and calls:

$$\begin{aligned}
\widehat{\mathsf{Entity}} &= (\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \mid (\mathsf{Stmt} \times \widehat{\mathsf{CallCtx}}) \mid (\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \\
&\quad \mid (\mathsf{Method} \times \widehat{\mathsf{CallCtx}}) \mid (\mathsf{Call} \times \widehat{\mathsf{CallCtx}}) \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{Val}}] &= \bot \mid \widehat{\mathsf{Obj}} \\
\widehat{\mathsf{Property}}[\kappa_{\mathsf{CallTarget}}] &= \widehat{\mathsf{CallTarget}} \\
\widehat{\mathsf{Obj}} &= \widehat{\mathsf{Obj}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}})) \\
\widehat{\mathsf{CallTarget}} &= \widehat{\mathsf{CallTarget}}(\mathcal{P}(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}} \times \mathsf{Method} \times \mathsf{Expr}^*))
\end{aligned}$$

Each entity is paired with a call context or heap context, which allows to tune the precision of the analysis. The static modules communicate via two kinds: Kind κ_{Val} refers to possible values of expressions and fields and the return value of methods. Values are abstract objects containing information about where objects were allocated. Kind $\kappa_{\mathsf{CallTarget}}$ refers to possible targets of method calls. Call targets are sets of receiver objects paired with the target method and their arguments.

To illustrate the analysis, Listing 1.2 shows the code of modules $\widehat{\mathsf{virtualCall}}$ and $\widehat{\mathsf{invokeReturn}}$. They implicitly communicate with each other via the store but do not call each other directly. Module $\widehat{\mathsf{virtualCall}}$ resolves virtual method calls by first fetching the points-to set of the receiver reference from the store. Afterwards, it iterates over all possible receivers and fetches possible target methods from the class table. Finally, it writes the new call target to the store. Storing the receiver object and argument expressions as part of the call target allows to reuse module $\widehat{\mathsf{invokeReturn}}$ for different types of calls. If the entity is a `Call` expression, module $\widehat{\mathsf{invokeReturn}}$ first fetches the targets of the call from the store. Then, it iterates over all targets, extends the call context with function

```
$\widehat{\mathsf{virtualCall}}$(e, $\widehat{\sigma}$) = e match
  case (call@Call(receiver, methodName, args), callCtx) =>
    receiverVal = $\widehat{\sigma}$((receiver, callCtx), $\kappa_{\mathsf{Val}}$)
    $\widehat{\mathsf{forObj}}$(receiverVal, $\widehat{\sigma}$) { (class, heapCtx, $\widehat{\sigma}'$) =>
      method = classTable(class, methodName)
      $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{CallTarget}} \mapsto \widehat{\mathsf{newCallTarget}}(\mathtt{class}, \mathtt{heapCtx}, \mathtt{method}, \mathtt{args})]$
    }
  case _ => $\widehat{\sigma}$

$\widehat{\mathsf{invokeReturn}}$(e, $\widehat{\sigma}$) = e match
  case (call@Call(_,_,_), callCtx) =>
    targets = $\widehat{\sigma}((\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{CallTarget}})$
    $\widehat{\mathsf{forCallTarget}}$(targets,$\widehat{\sigma}$){(class,heapCtx,method,args,$\widehat{\sigma}'$) => method match
      case SourceMethod(_,params,_) =>
        newCallCtx = $\widehat{\mathsf{extendCtx}}$(call.label,heapCtx,callCtx)
        $\widehat{\sigma}' \sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
          $\sqcup [(\mathtt{p}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}'((\mathtt{a}, \mathtt{callCtx}), \kappa_{\mathsf{Val}}) \mid (\mathtt{p}, \mathtt{a}) \in \mathsf{zip}(\mathtt{params}, \mathtt{args})]$
          $\sqcup [(\mathsf{VarRef}(\text{"}\mathtt{this}\text{"}), \mathtt{newCallCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{newObj}}(\mathtt{class}, \mathtt{heapCtx})]$
          $\sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\mathsf{nullPointer}}()]$
          $\sqcup [(\mathtt{call}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}((\mathtt{method}, \mathtt{newCallCtx}), \kappa_{\mathsf{Val}})]$
      case NativeMethod(_,_,_) => $\widehat{\sigma}'$
    }
  case Return(method,expr) =>
      $\widehat{\sigma} \sqcup [(\mathtt{method}, \mathtt{callCtx}), \kappa_{\mathsf{Val}} \mapsto \widehat{\sigma}(\mathtt{expr}, \mathtt{callCtx}, \kappa_{\mathsf{Val}})]$
  case _ => $\widehat{\sigma}$
```

Listing 1.2: Static modules for invoking calls and resolving virtual calls.

$\widehat{\mathsf{extendCtx}}$, binds the parameters to the values of the arguments and variable `this` to the receiver object. Furthermore, it registers the called method as an entity in the store, which in turn triggers module $\widehat{\mathtt{method}}$ to process the statements of the called method. Lastly, module $\widehat{\mathsf{invokeReturn}}$ writes the return value of a method to the method entity in the store and copies it to call entities of this method.

The modules depend on interface $\widehat{\mathsf{Objects}}$ shown in Listing 1.1 and an analogous interface for call targets. Operations $\widehat{\mathsf{newObj}}$ and $\widehat{\mathsf{newCallTarget}}$ create new abstract objects and call targets. Operations $\widehat{\mathsf{forObj}}$ and $\widehat{\mathsf{forCallTarget}}$ iterate over all objects and call targets. Interface $\widehat{\mathsf{Objects}}$ also includes an operation $\widehat{\mathsf{nullPointer}}$ not shown in the listing, which returns an empty set of object allocation-sites ($\widehat{\mathsf{Obj}}(\varnothing)$). The dynamic instances are analogous except that they operate on singleton types.

The dynamic modules compute a program's *heap* and describe its changes during execution. They are analogous to their static counterparts except that they operate on singleton types $\mathsf{Obj}(\mathsf{Class} \times \mathsf{HeapCtx})$ and $\mathsf{CallTarget}(\mathsf{Class} \times \mathsf{HeapCtx} \times \mathsf{Method} \times \mathsf{Expr}^*)$.

All dynamic modules combined do not cover the entire language. In particular, there are no dynamic modules that cover reflective calls. This means, as of now, the dynamic semantics of reflection is undefined, and the soundness proof

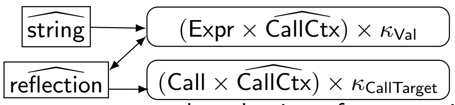

$\widehat{\text{string}}$
$\widehat{\text{reflection}}$
$(\text{Expr} \times \widehat{\text{CallCtx}}) \times \kappa_{\text{Val}}$
$(\text{Call} \times \widehat{\text{CallCtx}}) \times \kappa_{\text{CallTarget}}$
Arrows represent reads and writes of store entries

Fig. 2: Reflection Static Modules

only covers programs without reflective calls. We address this point with the following case study.

Reflection Analysis Reflection is a language feature that allows to query information about classes and methods at runtime [35]. Our language supports three reflective methods: Methods Class.forName and Class.getMethod retrieve classes and methods by a string, respectively. Method.invoke invokes a method, where the target method is determined at runtime. Reflection is notoriously difficult to statically analyze soundly and precisely [30]: analyses need to approximate the content of the string passed into a reflective call. If the analysis cannot determine the string precisely, it needs to overapproximate or risk unsoundness. In this case study, we choose the former to be able to prove the analysis sound.

This case study demonstrates two important features of our formalization: First, the reflection analysis reuses all pointer and call-graph modules of the previous section ($\widehat{\texttt{pointsTo}}$, $\widehat{\texttt{method}}$, $\widehat{\texttt{virtualCall}}$, and $\widehat{\texttt{invokeReturn}}$). It extends the value lattice to propagate new types of analysis information about strings. Even though the pointer analysis propagates new information, it does not require any changes and its soundness proof remains valid (Section 4.2). Second, the reflection analysis *cooperates* with the call-graph static module $\widehat{\texttt{virtualCall}}$ as reflective calls are regular virtual calls. For example, a call m.invoke(...) where variable m is of type Method is first resolved by virtual call resolution and its target Method.invoke is then resolved by reflective call resolution. Thus, both analyses add elements to the same set of call targets but can be proven sound independently from each other (Section 3).

The reflection analysis extends the $\widehat{\texttt{Obj}}$ values of the pointer analysis with three new types of values—$\widehat{\text{Str}}$, $\widehat{\text{Class}}$, and $\widehat{\text{Method}}$—as a reduced product [9]:

$$\widehat{\text{Property}}[\kappa_{\text{Val}}] = \bot \mid (\widehat{\text{Obj}} \times \widehat{\text{Str}} \times \widehat{\text{Class}} \times \widehat{\text{Method}})$$
$$\widehat{\text{Str}} = \bot \mid \text{String} \mid \top$$
$$\widehat{\text{Class}} = \mathcal{P}(\text{Class}) \mid \top$$
$$\widehat{\text{Method}} = \mathcal{P}(\text{Method}) \mid \top$$

String values are approximated with a constant lattice. Class and method values are approximated with a finite set of classes/methods or $\top$. We reuse the modules of the pointer and call-graph analysis by implementing a new instance of interface Objects in Listing 1.1 for the new values. The new instance is similar to $\widehat{\text{AllocationSiteAndStrings}}$ and iterates over all allocation-site information in strings, class/method values, and other objects.

The reflection analysis adds two new modules to the existing analysis in Figure 1. The new modules and their dependencies are visualized in Figure 2.

```
$\widehat{\mathsf{reflection}}$($e$, $\widehat{\sigma}$) = $e$ match
  case (call@Call(receiver, method, _), callCtx) =>
    target = $\widehat{\sigma}$((call, callContext), $\kappa_{\mathsf{CallTarget}}$)
    $\mathsf{for}\widehat{\mathsf{CallTarget}}$(target, $\widehat{\sigma}$) { (_,heapCtx,method,args,$\widehat{\sigma}'$) =>
      method match
      case NativeMethod("invoke") => arguments match
          case (invokeReceiver :: invokeArgs) =>
            invokeRecVal = $\widehat{\sigma}'$((invokeReceiver, heapCtx), $\kappa_{\mathsf{Val}}$)
            methodVal = $\widehat{\sigma}'$((receiver, callContext), $\kappa_{\mathsf{Val}}$)
            reflectiveTarget = $\widehat{\mathsf{methodInvoke}}$(invokeRecVal, methodVal,
                invokeArgs)
            $\widehat{\sigma}' \sqcup [(\texttt{call}, \texttt{callCtx}), \kappa_{\mathsf{CallTarget}} \mapsto \texttt{reflectiveTarget}]$
      ... }
  case _ => $\widehat{\sigma}$

$\widehat{\mathsf{methodInvoke}}$(recv:$\widehat{\texttt{Value}}$,methodVal:$\widehat{\texttt{Value}}$,invokeArgs:$\mathsf{Expr}^*$) = methodVal match
  case (_,_,_,methods) => $\widehat{\mathsf{CallTarget}}$({(c,h,m,invokeArgs) |
       (c,h)$\in$recv, m $\in$ methods, m $\in$ classTable(c)})
  case (_,_,_,$\top$) => $\widehat{\mathsf{CallTarget}}$({ (c,h,m,invokeArgs) |
       (c,h) $\in$ recv, method $\in$ classTable(class) })
  case $\bot$ => $\bot$
```

Listing 1.3: Static modules and operations for reflection.

Module $\widehat{\mathsf{reflection}}$ analyzes reflective calls to Class.forName, Class.getMethod, and Method.invoke. Module $\widehat{\mathsf{string}}$ analyzes string literals and concatenation. Listing 1.3 shows an excerpt of module $\widehat{\mathsf{reflection}}$ for Method.invoke. Module $\widehat{\mathsf{reflection}}$ first fetches the targets of a call resolved by module $\widehat{\mathsf{virtualCall}}$. If the call target is the native method `invoke`, module $\widehat{\mathsf{reflection}}$ matches on the arguments of the virtual call to extract the receiver and arguments of the reflective call target. Finally, it calls operation $\widehat{\mathsf{methodInvoke}}$ which returns the set of call targets.

Operation $\widehat{\mathsf{methodInvoke}}$ is part of an interface for reflective calls. The interface contains two other operations for retrieving class names and methods. $\widehat{\mathsf{methodInvoke}}$ matches on the call receiver and the method value. If the method value contains a finite set of methods, the operation checks if the receiver class has these methods and adds them as call targets. If the method value contains $\top$, the operation adds all methods of the receiver class to the set of call targets. This over-approximates the dynamic module reflection where only one method is added as a call target.

The dynamic reflection modules are analogous except that different types of values are alternatives. In contrast to Section 5.1, the dynamic pointer and call-graph modules combined with the reflection and string modules now cover the entire language. Thus, the analysis is sound for all programs, even those using reflection.

Field and Object Immutability Analysis The analysis of this case study computes the immutability of objects and their fields inspired by a class and field immutability analysis by Roth et al. [41]. This information is useful for assessing the thread safety of programs, where multiple threads have access to the same objects.

This case study highlights two important features of our formalization. First, the core dynamic semantics of our language does not describe the immutability property. Therefore, we need to prove the static immutability analysis sound with respect to a dynamic immutability analysis. The case study demonstrates that the immutability concern can be encapsulated with analysis and dynamic modules, added modularly to the existing analysis and dynamic semantics, and reasoned about independently (Section 3). It is unclear how this can be achieved with a non-modular, monolithic analysis implementation. Second, the immutability analysis adds new types of entities and kinds to the store and reuses all modules of the pointer, call-graph, and reflection analysis. Even though the reused modules can be called with the new entities and have access to new kinds in the store, their soundness proofs remain valid (Section 4.1).

The immutability analysis adds objects ($\mathsf{Class} \times \mathsf{HeapCtx}$) to the types of entities and adds kinds κ_{Mut} and κ_{Assign} for their immutability and the assignability of their fields:

$$\widehat{\mathsf{Entity}}' = \widehat{\mathsf{Entity}} \mid (\mathsf{Class} \times \mathsf{HeapCtx})$$
$$\widehat{\mathsf{Property}}[\kappa_{\mathsf{Mut}}] = \widehat{\mathsf{TransitivelyImmutable}} \mid \widehat{\mathsf{NonTransitivelyImmutable}} \mid \widehat{\mathsf{Mutable}}$$
$$\widehat{\mathsf{Property}}[\kappa_{\mathsf{Assign}}] = \widehat{\mathsf{Assignable}} \mid \widehat{\mathsf{NonAssignable}}$$

$\widehat{\mathsf{Mutable}}$ describes objects whose fields are reassigned. $\widehat{\mathsf{NonTransitivelyImmutable}}$ describes objects whose fields are not reassigned, but some objects transitively reachable via fields are mutated. $\widehat{\mathsf{TransitivelyImmutable}}$ describes objects whose fields are not reassigned and no transitively reachable objects are mutated. κ_{Assign} uses two elements for reassigned and not reassigned fields.

The immutability analysis consists of three modules shown in Figure 3. Module $\widehat{\mathsf{fieldAssign}}$ sets fields `f` of objects `o` to $\widehat{\mathsf{Assignable}}$ for every assignment of the form `x.f = e`, where `x` may point to `o`. Module $\widehat{\mathsf{fieldMutability}}$ sets a field to $\widehat{\mathsf{Mutable}}$ if the field is assignable, to $\widehat{\mathsf{NonTransitivelyImmutable}}$ if it is non-assignable but one of the pointed-to objects is mutable, and to $\widehat{\mathsf{TransitivelyImmutable}}$ otherwise. Lastly, module $\widehat{\mathsf{objectMutability}}$ sets an object's immutability to the least upper bound of the immutability of all of its fields.

The dynamic modules are analogous except that they operate on concrete objects instead of abstract objects.

Demand-Driven Reaching-Definitions Analysis As a final case study, we developed a demand-driven intra-procedural reaching-definitions analysis for our object-oriented language. This case study demonstrates that our theory lifts a restriction of existing soundness theories for generic interpreters. In particular, our theory also applies to analyses that do not follow the program execution order.

The analysis computes which definitions of variables and fields reach a statement without being overwritten. The analysis is demand-driven, as it performs

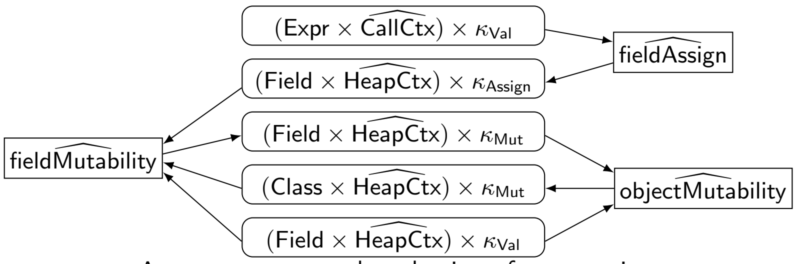

$(\mathsf{Expr} \times \widehat{\mathsf{CallCtx}}) \times \kappa_{\mathsf{Val}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Assign}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Mut}}$
$(\mathsf{Class} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Mut}}$
$(\mathsf{Field} \times \widehat{\mathsf{HeapCtx}}) \times \kappa_{\mathsf{Val}}$
$\widehat{\mathsf{fieldAssign}}$
$\widehat{\mathsf{fieldMutability}}$
$\widehat{\mathsf{objectMutability}}$
Arrows represent reads and writes of store entries

Fig. 3: Immutability Static Modules

the minimum amount of work to compute the reaching definitions of a query statement: the analysis only computes the reaching definitions of the query statement and its predecessors. Also, the analysis does not compute the entire control-flow graph, but only the query statement's predecessors.

The analysis consists of two modules $\widehat{\mathsf{reachingDefs}}$ and $\widehat{\mathsf{controlFlow}}$ similar to these discussed in Section 2. Module $\widehat{\mathsf{controlFlow}}$ calculates the set of control-flow predecessors of a given statement by computing the set of control-flow exits of the preceding statement within the abstract syntax tree. For example, the control-flow exits of an `if` statement are the exits of the last statements of both branches. The dynamic module controlFlow computes the predecessor immediately executed before the given statement. To this end, the module remembers the most recently executed statement in a mutable variable and only updates it if the given statement is the control-flow successor.

The main challenge in this case study was to find a dynamic module controlFlow that closely corresponds to the static module and still computes the correct control-flow predecessor. With a suitable dynamic module, the soundness proof of the static module became easier. Furthermore, we validated the correctness of the dynamic module with several unit tests.

5.2 Soundness Proofs of the Case Studies

We apply our theory to compositionally prove the analyses from the previous section sound. The proofs can be found in the supplementary material accompanying this paper. They are pen-and-paper proofs and do not make use of mechanization; but due to modularization, they are small and easy to verify.

Proving each analysis sound includes (a) proving each of its modules sound (Definition 8), (b) proving the instances of the property interface sound, and (c) verifying that Theorem 2 applies. To ensure the latter, we checked that there are no dependencies between modules and that all communication between them happens via the store (Definition 1). This can be easily checked by inspecting the code of the modules. Furthermore, we verified that modules do not make any assumption about abstract domains and are polymorphic in the store (Definition 7). This can be easily checked by inspecting the polymorphic type of the modules.

To prove the individual modules of an analysis sound, step (a) in the overall soundness proof, we use two techniques. The first uses the observation that static modules and their corresponding dynamic modules are often very similar, except for the types of entities and properties. We can abstract over these differences with a generic module, from which we derive both a dynamic and static module. Then, soundness follows immediately as a free theorem from parametricity [28]. In cases where abstracting with a generic module is not possible or desirable, we resort to a manual proof. We were able to use the first technique for all modules, except for $\widehat{\mathsf{method}}$, $\widehat{\mathsf{reachingDefs}}$, and $\widehat{\mathsf{controlFlow}}$. For illustrating cases where we need manual proofs, consider the flow-insensitive static module $\widehat{\mathsf{method}}$ of the pointer analysis and its corresponding dynamic module method. While we could potentially derive them from the same generic module, the derived static module would be less performant, because it would trigger the analysis of parts of the code, e.g., `if` conditions, which our current flow-insensitive module does not. This is an example where our approach leads to more freedom in the design of static analyses than the existing approach based on a generic interpreter (Section 6.1).

The soundness proofs of the static modules are reusable across different analyses, because the modules can be soundly lifted to supersets of entities and kinds (Lemma 1). For example, the immutability analysis adds class entities, requiring to lift the modules of the pointer and reflection analysis. Furthermore, the soundness proofs of static modules can be reused because the proofs are independent of the lattices used (Definition 8). For example, the reflection analysis reuses all modules of the pointer analysis, extending the value lattice with string, class, and method information. The soundness proofs of the pointer static modules remain valid because they do not depend on a specific value lattice. Instead, the proofs of the pointer modules depend on soundness lemmas of the newObj and forObj operations of $\mathsf{Objects}$ interface.

Finally, we consider step (b) in the overall soundness proof – the soundness proof of the instances of the property interface. These instances need to be proven sound manually, as the proof cannot be decomposed any further. To prove them sound, we proved each of their operations sound. For the pointer analysis we needed to prove 7 operations sound, for the reflection analysis 6 operations, for the immutability analysis 6 operations, and for the reaching-definitions analysis 0 operations. Of these 19 operations, 13 could be proven sound trivially, requiring only a single proof step after unfolding the definitions. The remaining 6 operations required more elaborate proofs with multiple steps and case distinctions. These include $\widehat{\mathsf{forObj}}$ from the pointer analysis, $\widehat{\mathsf{classForName}}$, $\widehat{\mathsf{getMethod}}$, and $\widehat{\mathsf{methodInvoke}}$ from the reflection analysis, and $\widehat{\mathsf{getFieldMutability}}$ and $\widehat{\mathsf{joinMutability}}$ from the immutability analysis.

6 Related Work

In this section, we discuss work related to compositional and reusable soundness proofs as well as to modular analysis architectures.

6.1 Theories for Compositional and Reusable Soundness Proofs

All works discussed in this subsection, including our own, build upon the theory of *abstract interpretation.* Abstract interpretation is a formal theory of sound static analyses, first conceived by Cousot et al. [12] but since then has found wide spread adoption in academia and industry [13, 16, 22, 25, 33, 44]. Abstract interpretation defines soundness of static analyses but does not explain how soundness can be proved. As we elaborate in the introduction, soundness proofs of practical analyses for real-world languages are difficult because they relate two complicated semantics often described in different styles. Proof attempts of such analyses often fail due to high proof complexity and effort. Furthermore, existing proofs are prone to become invalid if the static or dynamic semantics change and reestablishing proofs is laborious and complicated.

Domain constructions, such as *reduced products* and *reduced cardinal powers* [12], combine multiple existing abstract domains to improve their precision. They can be used to compose the soundness proof of operations on the abstract domain, e.g, primitive arithmetic, boolean, or string operations. However, they cannot be used to compose the soundness proof of the analysis of statements, e.g., assignments, loops, or procedure calls. In contrast, the blackboard architecture is capable to compose soundness proofs of both of these types of operations.

Darais et al. [14] developed a theory for soundness proofs, in which the static and dynamic semantics are derived from a *small-step generic interpreter* that describes the operational semantics of the language without mentioning details of static or dynamic semantics. The small-step generic interpreter is instantiated with reusable *Galois transformers* that capture aspects such as flow- or path-sensitivity and allow to change an existing analysis while preserving soundness. Galois transformers can be proven sound once and for all and their soundness proofs are reusable across different analyses. However, the approach does not compose soundness proofs of static semantics derived from the generic interpreter.

Keidel et al. [28] developed a theory for *big-step abstract interpreters*, deriving both the static and dynamic semantics from a generic big-step interpreter. The theory enables soundness composition [28, Theorem 4 and 5] if the generic interpreter is implemented with *arrows* [23] or in a meta-language which enjoys *parametricity.* But there is no theory how parts of soundness proofs can be reused between different analyses. Keidel et al. [27] later refined the theory by introducing reusable *analysis components* that capture different aspects of the language such as values, mutable state, or exceptions and are described with *arrow transformers* [23]. While components can be proven sound independently from each other, their *composition* requires glue code, which needs to be proven sound. Furthermore, the composition creates large arrow transformer stacks – that, unless optimized away by the compiler, may lead to inefficient analysis code. For example, a taint analysis for WebAssembly developed by using the approach depends on a stack of 18 arrow transformers.Eliminating the overhead of an arrow transformer stack of this size requires aggressive inlining and optimizations causing binary bloat and excessive compile times.

Bodin et al. [5] developed a theory of compositional soundness proofs for a style of semantics called *skeletal semantics*, which consists of hooks (recursive calls to the interpreter), filters (tests if variables satisfy a condition), and branches. The dynamic and static semantics are derived from the same *skeleton.* Also, soundness of the instantiated skeleton follows from soundness of the dynamic and static instance [5, Lemma 3.4 and 3.5]. However, their work does not describe how proofs can be reused across different analyses.

To recap, in all theories above the static and dynamic semantics *must* be derived from the same generic interpreter. This restricts what types of analyses can be derived. In particular, the static analysis must closely follow the program execution order dictated by the generic interpreter and it is unclear how static analyses can be derived that do not closely follow the program execution order. For example, backward analyses process programs in reverse order, flow-insensitive analyses may process statements in any order, and summary-based analyses construct summaries in bottom-up order. Our work lifts the restriction that static and dynamic semantics must be derived from the same artifact. static modules and corresponding dynamic modules must follow the blackboard architecture style, but else do not need to share any commonalities. This gives greater freedom as to which types of analyses can be implemented. For example, the blackboard analysis architecture has been used in prior work to develop backward analyses [17], on-demand/lazy analyses [19, 41], and summary-based analyses [21]. We also demonstrated in Section 5.1 that our theory applies to a demand-driven reaching definitions analysis. It is unclear how such an analysis can be derived from a generic interpreter.

6.2 Modular Analysis Architectures

These architectures describe how to implement static analyses modularly. Modular analysis architectures are a necessary requirement to develop theories for compositional and reusable soundness proofs. The theories give formal guarantees about proof independence, composition, and reuse.

Our work formally defines the blackboard analysis architecture used in the *OPAL framework* [15,21]. In the past, OPAL has been used to implement state-of-the-art analyses for method purity [19], class- and field-immutability [41], and call-graphs [40] for Java Virtual Machine bytecode. Furthermore, OPAL features escape analyses and a solver for IFDS analyses [21] as well as a fixpoint algorithm that parallelizes the analysis execution [20].

Prior to the work presented in this paper, no formalization of the blackboard analysis architecture and no theory for its soundness existed. Our formalization captures the core of the OPAL framework, while deliberately ignoring implementation details. For example, our formalization does not describe the fixpoint algorithm and the order in which it executes static modules to resolve their dependencies. Proving the fixpoint algorithm correct is a separate concern compared to proving analyses sound, which is the focus of our formalization. That said, our formalization covers a variety of OPAL's features described by Helm et al. [21]. For example, OPAL supports *default* and *fallback* properties for missing properties in the store. Fallback properties can be described by our formalization by adding them to the initial store passed to the fixpoint algorithm. We deliberately leave out default properties, which are an edge case in OPAL to mark properties not computed, e.g., because of dead code. They could be added to our formalization by extending analyses with a second set of static modules to be executed after the fixpoint is reached. Furthermore, OPAL supports *optimistic* analyses which ascend the lattice and *pessimistic* analyses which descend the lattice during fixpoint iteration. Both of these are covered by our formalization which describes analyses as monotone functions that ascend or descend the lattice. However, we deliberately do not cover OPAL's mechanisms for allowing interaction between optimistic and pessimistic analyses, another edge case.

Configurable program analysis (CPA) [4] is a modular analysis architecture that describes analyses with transfer relation between control-flow nodes. CPAs can be systematically composed with reduced products. Furthermore, soundness of a component-wise transfer relation follows directly from soundness of its constituents. However, it is unclear how soundness proofs of primitive CPAs can be composed or how proof parts can be reused across analyses.

Doop [7] is a framework which describes analysis with relations in Datalog. Each relation is defined as a set of rules. These rules can be modularly added or replaced, without requiring changes to other rules. While individual analyses in Doop have been proven sound [43], the proofs are not compositional or reusable. In particular, if one rule changes, the proof becomes invalid and needs to be reestablished. This is because the proof reasons about soundness of all rules at once instead of individual rules or relations. The *IncA* framework [45] also describes analyses in Datalog, but allows relations over lattices instead of only sets. However, no soundness theory for its analyses exists. Similar to IncA, the *Flix* framework [37] describes analyses with lattice-based Datalog relations and functions. Flix proves individual functions sound with an automated theorem prover [36]. While an automated theorem prover reduces the proof effort and increases proof trustworthiness, there is no guarantee that the automated theorem prover is able to conduct a proof. Furthermore, the automated theorem prover does not establish a soundness proof of Datalog relations.

Verasco [26] is a modular analysis for *C#minor* [32], an intermediate language used by the *CompCert* C compiler [33]. Verasco is proven sound with the *Coq* proof assistant [3]. The soundness proof of the abstract C#minor semantics is independent of the abstract domain, which makes the proof reusable for other abstract domains. However, the abstract semantics is proven sound w.r.t. the *standard* concrete semantics. Thus, the proof cannot be reused for abstract semantics which approximate *non-standard* concrete semantics, such as information flow analyses [2] or liveness analyses [11].

Several other modular analysis architectures [24, 31, 42] do not have formal theories for soundness.

6.3 Monolithic Soundness Proofs

In this subsection, we compare compositional and reusable soundness proof theories to ad-hoc monolithic proofs and discuss their trade-offs.

Monolithic soundness proofs consider the entire analysis and dynamic semantics as a whole. This complicates the proof because there is no separation of concerns to manage the complexity of modern programming languages. Furthermore, monolithic soundness proofs are harder to maintain. In particular, whenever the analysis needs to be updated to support a new version of the language, or whenever the analysis is fine-tuned to improve precision and scalability, the soundness proof becomes invalid and needs to be reestablished. However, reestablishing the soundness proof is difficult because it is unclear which parts of the proof have become invalid and need to be updated. In contrast, compositional soundness proofs narrow the proofscope to individual modules, which decreases the proofs' complexity. Furthermore, compositional soundness proofs are easier to maintain as changes to individual modules only invalidate their particular soundness proof, while the proofs of other modules remain valid.

The main benefit of monolithic soundness proofs over compositional proofs is that analyses may be proven sound with respect to existing formal dynamic semantics.However, often no suitable formal dynamic semantics exists and analyses still have to be proven sound with respect to customly defined or modified dynamic semantics. For example, *HornDroid* [8] is proven sound with respect to a custom instrumented JVM small-step semantics and *Jaam*[6] is proven sound with respect to a custom JVM semantics in form of an abstract machine [22]. Furthermore, analyses of properties not present in standard language semantics need to be proven sound with respect to instrumented dynamic semantics. For example, a static taint analysis needs to be proven sound with respect to an instrumented dynamic semantics with taint information. In contrast, compositional soundness proofs require a one-time cost of formalizing a modular dynamic semantics for a language. Once this is done, several analyses can be proven sound with respect to this dynamic semantics. Furthermore, the dynamic semantics can be modularly extended to describe new aspects such as taint information.

7 Future Work

In this section, we discuss limitations of our work and how these limitations can be addressed in the future.

First, our soundness theory requires that static analyses and dynamic semantics are described in the blackboard analysis architecture. It is unclear how easily existing analyses and dynamic semantics be adapted to the architecture. In Section 2.2, we showed how existing small-step dynamic semantics can be described as a module and Helm et al. [21] implemented a wide range of static analyses in the architecture. In the future, we want to investigate how other styles of static and dynamic semantics can be adapted to the architecture.

[6] `https://github.com/Ucombinator/jaam`

Second, our soundness theory requires that all static modules are sound. However, in practice static analyses are deliberately unsound due to complicated language features [34]. In the future, we want to investigate how the blackboard analysis architecture can be used to localize unsoundness. Specifically, unsound analysis results could be tagged with the name of the module that produced them. All results derived from unsound results then propagate the tags. This way, it is always clear which results are potentially unsound and which modules caused unsoundness.

Lastly, our work has focused on soundness, i.e., analyses do not produce false-negative results. A complementary property to soundness is completeness, i.e., analyses do not produce false-positives results. No false-positive results are especially important if analyses produce warnings that are to be inspected by developers. In the future, we want to investigate if our theory can be extended to prove completeness of static analyses.

8 Conclusion

In this work, we developed a theory for compositional and reusable soundness proofs for static analyses in the blackboard analysis architecture. The blackboard analysis architecture modularizes the implementation of static analyses with analyses composed of independent static modules. We proved that soundness of an analysis follows directly from independent soundness proofs of each module. Furthermore, we extended our theory to enable the reuse of soundness proofs of existing modules across different analyses. We evaluated our approach by implementing four analyses and proving them sound: A pointer, a call-graph, a reflection, an immutability analysis, and a demand-driven reaching definitions analysis.

9 Data Availability

The implementation of the case studies and proofs are provided as an artifact available at `https://doi.org/10.5281/zenodo.10418484`.

Acknowledgments. This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

1. Afonso, V.M., de Geus, P.L., Bianchi, A., Fratantonio, Y., Kruegel, C., Vigna, G., Doupé, A., Polino, M.: Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016)

2. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. 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. 874–887. ACM (2017). https://doi.org/10.1145/3009837.3009889
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
5. Bodin, M., Gardner, P., Jensen, T.P., Schmitt, A.: Skeletal semantics and their interpretations. Proc. ACM Program. Lang. **3**(POPL), 44:1–44:31 (2019). https://doi.org/10.1145/3290357
6. Bogdanas, D., Rosu, G.: K-java: A complete semantics of java. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
7. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Arora, S., Leavens, G.T. (eds.) Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA. pp. 243–262. ACM (2009). https://doi.org/10.1145/1640089.1640108
8. Calzavara, S., Grishchenko, I., Maffei, M.: Horndroid: Practical and sound static analysis of android applications by SMT solving. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016. pp. 47–62. IEEE (2016). https://doi.org/10.1109/EuroSP.2016.16
9. Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. In: Banerjee, A., Danvy, O., Doh, K., Hatcliff, J. (eds.) Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013. EPTCS, vol. 129, pp. 325–336 (2013). https://doi.org/10.4204/EPTCS.129.19
10. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics **81**(1), 43–57 (1979). https://doi.org/10.2140/pjm.1979.82.43
11. Cousot, P.: Syntactic and semantic soundness of structural dataflow analysis. In: Chang, B.E. (ed.) Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11822, pp. 96–117. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_6
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas,

USA, January 1979. pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778

13. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Sagiv, S. (ed.) Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3
14. Darais, D., Might, M., Horn, D.V.: Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 552–571. ACM (2015). https://doi.org/10.1145/2814270.2814308
15. Eichberg, M., Hermann, B.: A software product line for static analyses: the OPAL framework. In: Arzt, S., Santelices, R.A. (eds.) Proceedings of the 3rd ACM SIGPLAN International Workshop on the State Of the Art in Java Program analysis, SOAP 2014, Edinburgh, UK, Co-located with PLDI 2014, June 12, 2014. pp. 2:1–2:6. ACM (2014). https://doi.org/10.1145/2614628.2614630
16. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. pp. 3–18. IEEE Computer Society (2018). https://doi.org/10.1109/SP.2018.00058
17. Glanz, L., Müller, P., Baumgärtner, L., Reif, M., Amann, S., Anthonysamy, P., Mezini, M.: Hidden in plain sight: Obfuscated strings threatening your privacy. In: Sun, H., Shieh, S., Gu, G., Ateniese, G. (eds.) ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020. pp. 694–707. ACM (2020). https://doi.org/10.1145/3320269.3384745
18. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with webassembly. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 185–200. ACM (2017). https://doi.org/10.1145/3062341.3062363
19. Helm, D., Kübler, F., Eichberg, M., Reif, M., Mezini, M.: A unified lattice model and framework for purity analyses. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 340–350. ACM (2018). https://doi.org/10.1145/3238147.3238226
20. Helm, D., Kübler, F., Kölzer, J.T., Haller, P., Eichberg, M., Salvaneschi, G., Mezini, M.: A programming model for semi-implicit parallelization of static analyses. In: Khurshid, S., Pasareanu, C.S. (eds.) ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020. pp. 428–439. ACM (2020). https://doi.org/10.1145/3395363.3397367
21. Helm, D., Kübler, F., Reif, M., Eichberg, M., Mezini, M.: Modular collaborative program analysis in OPAL. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event,

USA, November 8-13, 2020. pp. 184–196. ACM (2020). https://doi.org/10.1145/3368089.3409765

22. Horn, D.V., Might, M.: Abstracting abstract machines. 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. 51–62. ACM (2010). https://doi.org/10.1145/1863543.1863553
23. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1-3), 67–111 (2000). https://doi.org/10.1016/S0167-6423(99)00023-4
24. Johnson, N.P., Fix, J., Beard, S.R., Oh, T., Jablin, T.B., August, D.I.: A collaborative dependence analysis framework. In: Reddi, V.J., Smith, A., Tang, L. (eds.) Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017. pp. 148–159. ACM (2017)
25. Jourdan, J.: Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié). Ph.D. thesis, Paris Diderot University, France (2016)
26. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 247–259. ACM (2015). https://doi.org/10.1145/2676726.2676966
27. Keidel, S., Erdweg, S.: Sound and reusable components for abstract interpretation. Proc. ACM Program. Lang. **3**(OOPSLA), 176:1–176:28 (2019). https://doi.org/10.1145/3360602
28. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. Proc. ACM Program. Lang. **2**(ICFP), 72:1–72:26 (2018). https://doi.org/10.1145/3236767
29. Kester, D., Mwebesa, M., Bradbury, J.S.: How good is static analysis at finding concurrency bugs? In: Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010. pp. 115–124. IEEE Computer Society (2010). https://doi.org/10.1109/SCAM.2010.26
30. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection: literature review and empirical study. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 507–518. IEEE / ACM (2017). https://doi.org/10.1109/ICSE.2017.53
31. Lerner, S., Grove, D., Chambers, C.: Composing dataflow analyses and transformations. In: Launchbury, J., Mitchell, J.C. (eds.) Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002. pp. 270–282. ACM (2002). https://doi.org/10.1145/503272.503298
32. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (dec 2009). https://doi.org/10.1007/s10817-009-9155-4, https://doi.org/10.1007/s10817-009-9155-4
33. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert - A Formally Verified Optimizing Compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. SEE, Toulouse, France (Jan 2016)

34. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015). https://doi.org/10.1145/2644805
35. Livshits, V.B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 139–160. Springer (2005). https://doi.org/10.1007/11575467_11
36. Madsen, M., Lhoták, O.: Safe and sound program analysis with flix. In: Tip, F., Bodden, E. (eds.) Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. pp. 38–48. ACM (2018). https://doi.org/10.1145/3213846.3213847
37. Madsen, M., Yee, M., Lhoták, O.: From datalog to flix: a declarative language for fixed points on lattices. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 194–208. ACM (2016). https://doi.org/10.1145/2908080.2908096
38. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6
39. Nii, H.P.: Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. AI Mag. **7**(2), 38–53 (1986)
40. Reif, M., Kübler, F., Eichberg, M., Helm, D., Mezini, M.: Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. pp. 251–261. ACM (2019). https://doi.org/10.1145/3293882.3330555
41. Roth, T., Helm, D., Reif, M., Mezini, M.: Cifi: Versatile analysis of class and field immutability. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. pp. 979–990. IEEE (2021). https://doi.org/10.1109/ASE51524.2021.9678903
42. Schubert, P.D., Leer, R., Hermann, B., Bodden, E.: Into the woods: Experiences from building a dataflow analysis framework for C/C++. In: 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg, September 27-28, 2021. pp. 18–23. IEEE (2021). https://doi.org/10.1109/SCAM52516.2021.00011
43. Smaragdakis, Y., Kastrinis, G.: Defensive points-to analysis: Effective soundness via laziness. In: Millstein, T.D. (ed.) 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands. LIPIcs, vol. 109, pp. 23:1–23:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.23
44. Stein, B., Chang, B.E., Sridharan, M.: Demanded abstract interpretation. 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. 282–295. ACM (2021). https://doi.org/10.1145/3453483.3454044
45. Szabó, T., Bergmann, G., Erdweg, S., Voelter, M.: Incrementalizing lattice-based program analyses in datalog. Proc. ACM Program. Lang. **2**(OOPSLA), 139:1–139:29 (2018). https://doi.org/10.1145/3276509

46. Taneja, J., Liu, Z., Regehr, J.: Testing static analyses for precision and soundness. In: CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020. pp. 81–93. ACM (2020). https://doi.org/10.1145/3368826.3377927

Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation*

Pierre Lermusiaux and Benoît Montagu(✉)

Inria, Campus universitaire de Beaulieu, Rennes, France
`pierre.lermusiaux@inria.fr`, `benoit.montagu@inria.fr`

Abstract. Exception handling is a key feature in modern programming languages. Exceptions can be used to deal with errors, or as a means to control the flow of execution of a program. Since they might unexpectedly terminate a program, unhandled exceptions are a serious safety concern. We propose a static analysis to detect uncaught exceptions in functional programs, that is defined as an abstract interpreter. It computes a description of the values potentially returned by a program using a novel abstract domain, that can express inductively defined sets of values. Simultaneously, the analysis infers the possibly raised exceptions, by computing in the *abstract exception monad*. This abstract interpreter has been implemented as an effective static analyser for a large subset of OCaml programs, that supports mutable data types, the OCaml module system, and dynamically extensible data types such as the exception type. The analyser has been evaluated on several hundreds of OCaml programs.

Keywords: Static Analysis · Exceptions · Higher-Order Programs · Abstract Interpretation · Abstract Domain for Trees

1 Introduction

Programs that run in critical environments need to comply with strong safety guarantees. The minimal guarantee one expects for critical software is the *absence of runtime failures*. Sound static analyses can provide such guarantees statically, for every possible execution of a program, and in a fully automatic manner.

The static typing discipline found in the ML family of languages is such a static analysis technique, that brought strong safety guarantees to programs at a very low cost: *well-typed programs cannot "go wrong"* [48]. This soundness theorem for well-typed ML programs, however, does not preclude programs from abruptly ending with uncaught exceptions. Several analyses for ML-like languages have been developed to detect such undesirable behaviours, that were either leveraging type and effect systems [38,54], or that were based on variants of control-flow analyses or set constraints [68,67,14,15,66]. The recent success of *algebraic effects* and their introduction in popular languages such as OCaml [37] has renewed the interest in the static detection of uncaught exceptions and effects.

* This work was funded by the Salto grant, supported by Nomadic Labs and Inria.

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 391–420, 2024.
https://doi.org/10.1007/978-3-031-57267-8_15

Analysing uncaught exceptions in ML is a difficult problem, because data flow and control flow are interdependent. This is not only due to the first class nature of functions, but also due to the first class nature of exceptions themselves, *e.g.*, they can be taken as parameters, recorded in data structures or in mutable references. Furthermore, exceptions can carry any value as argument—including functions—and new exceptions can be dynamically generated at runtime.

In this paper, we propose a static analysis for a higher-order language, in which exceptions are first-class values. The analysis is based on the abstract interpretation framework [9]. It is a forward value analysis that infers which values any program point can compute, and which exceptions they might raise. For this purpose, we introduce a novel abstract domain that can represent recursively defined sets of values. We define a *widening* operator for this abstract domain, that is responsible for finding recursive generalisations of solutions.

Our analysis leverages this abstract domain to represent both possible values and exceptions, thanks to the *abstract exception monad*. This monad—that can also be used as an abstract domain—is an abstraction of the exception monad, that collects all values and exceptions.

We define our analysis as a big-step monadic interpreter, written in the *open recursive style*, that was emphasised in the "Abstracting Definitional Interpreter" approach [11]. Then, we obtain an effective analyser by applying a generic, dynamic fixpoint solver [6,63,59,24,12,30]. We prove that our analysis is sound, under the soundness assumption of the fixpoint solver.

We extend the analysis to handle a large subset of the OCaml language. In particular, it supports the dynamic creation of exceptions, mutable state, modules and functors. The analysis is so far limited to *sequential* programs that do not perform system calls, do not use the Gc or Obj modules, and do not employ recursive modules, general recursive definitions of values, objects, classes, arrays, or floats. We implemented an OCaml prototype for this analyser. It reports the possibly thrown exceptions and an over-approximation of the data they carry, along with an abstraction of the call trace that led to the program point where the exception was raised. We discuss some implementation choices, and evaluate the precision and performance of our analyser on 290 programs, that include examples from the literature and from the OCaml compiler's test suite.

2 Overview

Let us consider the classic example of the factorial function, as written below in a *continuation passing style*.

```
let rec fact_cont n i k =
  if i >= n then k i else
  fact_cont n (i + 1) (fun x -> k (x * i))
let fact n = fact_cont n 1 (fun x -> x)
let result = fact 5
```

The `fact_cont` function recursively calls itself with increasing values of its parameter `i`, until the value `n` is reached.

We are interested in finding which values (and exceptions) this program might return. To answer this question, we first need to find the possible continuations the function `fact_cont` can be called with, and, importantly, we need an abstract domain in which we can express this set, or an over-approximation thereof.

With the abstract domain that we introduce in §4, we can express such a set as the following abstract value:

$$\mu\alpha.\, \{\mathsf{funs} : \{(\lambda x.\, x) \mapsto \{\};\ (\lambda x.\, k\ (x * i)) \mapsto \{i \mapsto \{\mathsf{ints} : [1, +\infty]\}; k \mapsto \alpha\}; \}\}$$

This abstract value represents a recursively-defined set—as indicated by the μ constructor—that is locally named α. This set is composed of function closures, that can be either the identity function, or the function $\lambda x.\, k\ (x * i)$, considered in an environment where the variable i is bound to an integer that is greater or equal to 1, and where the variable k is recursively bound to the local variable α, *i.e.*, to a value of the set we are defining.

Our abstract domain can also express structural invariants on data, such as the one for red-black trees [52], that forbids red nodes from having red children:

$$\mu\alpha. \left\{ \mathsf{constructs} : \left\{ \begin{array}{l} E : (); \\ R : \begin{pmatrix} \{\mathsf{constructs} : \{E : (), B : (\alpha, \{\mathsf{ints} : \top\}, \alpha)\}\}; \\ \{\mathsf{ints} : \top\}, \\ \{\mathsf{constructs} : \{E : (), B : (\alpha, \{\mathsf{ints} : \top\}, \alpha)\}\} \end{pmatrix}; \\ B : (\alpha, \{\mathsf{ints} : \top\}, \alpha) \end{array} \right\} \right\}$$

Our abstract domain bears a strong similarity with the theory of equi-recursive types [56], in the sense that *recursion* is a core aspect of our definition. However, it differs from recursive types, as function types are absent: sets of closures are used instead. Moreover, it is parameterised by a non-relational abstract domain used to represent integers values—which is not possible with *simple* type systems.

We leverage our abstract domain and define a static analysis for a call-by-value λ-calculus with pattern matching, exception handling, and first-class exceptions (§3). In this language, the order of evaluation is made explicit by let bindings, and pattern matching is *exhaustive* and *non-ambiguous* [8]. These requirements drastically simplify the semantics of programs and their analysis. The analysis is defined as an abstract interpreter that performs a forward value analysis (§5).

Based on this small abstract interpreter, we sketch (§6) several extensions that we implemented to obtain a static analyser for a subset of OCaml programs. The implementation uses an intermediate language that is close to the one of §3, into which we translated the OCaml typed abstract syntax tree. We evaluated the precision and performance of our analyser on 290 OCaml programs, written in a variety of styles (direct, CPS, monadic, *etc.*). We discuss these experimental results (§7), cover related work (§8), and finish with conclusive remarks (§9).

3 A λ-calculus With Exceptions

We introduce as an intermediate language a λ-calculus with pattern matching and exception handling. Its syntax resembles the monadic normal form, where the order of evaluation is made explicit with let bindings.

Definition 1. *Given $\mathcal{C}$ a set of constructor symbols, we give the following inductive definition of patterns p, q, and expressions t, u, r:*

$$\begin{array}{rl} p,\ q \in \mathbb{P} ::= & x \mid n \mid c(p_1, \dots, p_k) \mid p_1 + p_2 \mid p \setminus q \\ t,\ u,\ r \in \mathbb{T} ::= & x \mid n \mid x_1\ op\ x_2 \mid c(x_1, \dots, x_k) \\ & \mid \mu f.\, \lambda x.\, t \mid f\ y \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{raise}\ x \\ & \mid \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \\ & \mid \mathsf{dispatch}\ t\ \mathsf{with}\ \mathsf{val}\ x \Rightarrow u \mid \mathsf{exn}\ y \Rightarrow r \end{array}$$

where n is a constant integer, c is a constructor of $\mathcal{C}$, op is a binary operation on integers, and where the pattern q cannot contain any complement $p_1 \setminus p_2$.

We consider a pattern syntax and formalism inspired from [8]. The pattern disjunction $p+q$ matches any value matched by p or q, and the pattern complement $p \setminus q$ matches any value that is matched by p but not by q.

As in the OCaml typed AST, variables carry a type. We may write x_τ to denote that the variable x is of type τ. Patterns are linear, *i.e.*, sub-patterns of constructor patterns cannot share variables. All functions are recursive by default. If f does not occur in the expression t, then we write $\lambda x.\, t$ instead of $\mu f.\, \lambda x.\, t$.

The values of this language are integer constants, constructors applied to values, and function closures, that contain an environment of values:

$$\begin{array}{l} v \in \mathbb{V} ::= n \mid c(v_1, \dots, v_k) \mid \langle E, \mu f.\, \lambda x.\, t\rangle \quad \text{where } \operatorname{dom} E = \operatorname{fv}(\mu f.\, \lambda x.\, t) \\ E \in \mathbb{E} ::= [] \mid E, x \mapsto v \end{array}$$

Patterns induce a matching relation over values, that is described, with regard to a given environment E, by recursion on patterns:

$$\begin{array}{rcl@{\quad}c@{\quad}l} x & \prec\!\!\prec_E & v & \Longleftrightarrow & E(x) = v \\ c(p_1, \dots, p_n) & \prec\!\!\prec_E & c(v_1, \dots, v_n) & \Longleftrightarrow & \bigwedge_{i=1}^{n} p_i \prec\!\!\prec_E v_i \\ p + q & \prec\!\!\prec_E & v & \Longleftrightarrow & p \prec\!\!\prec_E v \ \vee\ q \prec\!\!\prec_E v \\ p \setminus q & \prec\!\!\prec_E & v & \Longleftrightarrow & p \prec\!\!\prec_E v \ \wedge\ q \not\prec\!\!\prec v \end{array}$$

We say that a pattern p matches a value v, denoted $p \prec\!\!\prec v$, iff there exists an environment E such that $p \prec\!\!\prec_E v$. In such case, we write $E\langle p \prec\!\!\prec v\rangle$ the smallest environment such that $p \prec\!\!\prec_{E\langle p \prec\!\!\prec v\rangle} v$.

Thanks to this pattern-matching formalism, we can focus on the class of programs where pattern matching is *exhaustive* and *non-ambiguous*, *i.e.*: In a term $\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$ where $t : \tau$, we require that for any value $v : \tau$, there exists a unique $1 \le i \le n$ such that $p_i \prec\!\!\prec v$. The work presented in [8] shows how to *disambiguate* patterns, *i.e.*, how to make any pattern match non-ambiguous. We restrict ourselves to non-ambiguous patterns, because it simplifies both the dynamic semantics and the analysis of programs.

We present in Figure 1 a *call-by-value* big-step semantics for our language. We write $t \Downarrow_{\mathsf{val}} v$ to denote that the expression term t reduces to the value v, and we write $t \Downarrow_{\mathsf{exn}} v$ to denote that the reduction of t *raises* an exception evaluated as v. In this language, any value can be raised as an exception. The evaluation rules are mostly standard. We briefly explain the rules for match and dispatch.

$$\frac{}{E \vdash x \Downarrow_{\mathsf{val}} E(x)}\ \textsc{Var} \qquad \frac{}{E \vdash n \Downarrow_{\mathsf{val}} n}\ \textsc{Int} \qquad \frac{}{E \vdash x_1\ op\ x_2 \Downarrow_{\mathsf{val}} E(x_1)\ [\![op]\!]\ E(x_2)}\ \textsc{Op}$$

$$\frac{}{E \vdash \mathsf{raise}\ x \Downarrow_{\mathsf{exn}} E(x)}\ \textsc{Raise} \qquad \frac{}{E \vdash c(x_1, \dots, x_k) \Downarrow_{\mathsf{val}} c(E(x_1), \dots, E(x_k))}\ \textsc{Const}$$

$$\frac{E' = E|_{\mathrm{fv}(\mu f.\,\lambda x.\,t)}}{E \vdash \mu f.\,\lambda x.\,t \Downarrow_{\mathsf{val}} \langle E', \mu f.\,\lambda x.\,t\rangle}\ \textsc{Lam} \qquad \frac{E(y) = \langle E', \mu f.\,\lambda x.\,t\rangle \quad E', f \mapsto E(y), x \mapsto E(z) \vdash t \Downarrow_m v}{E \vdash y\ z \Downarrow_m v}\ \textsc{App}$$

$$\frac{E \vdash t_1 \Downarrow_{\mathsf{val}} v_1 \qquad E, x \mapsto v_1 \vdash t_2 \Downarrow_m v_2}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_m v_2}\ \textsc{Let} \qquad \frac{E \vdash t_1 \Downarrow_{\mathsf{exn}} v}{E \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow_{\mathsf{exn}} v}\ \textsc{LetRaise}$$

$$\frac{E \vdash t \Downarrow_{\mathsf{val}} v \qquad p_i \prec\!\!\!\prec v \qquad E, E\langle p_i \prec\!\!\!\prec v\rangle \vdash u_i \Downarrow_m v' \qquad 1 \le i \le n}{E \vdash \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \Downarrow_m v'}\ \textsc{Match}$$

$$\frac{E \vdash t \Downarrow_{\mathsf{exn}} v}{E \vdash \mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n \Downarrow_{\mathsf{exn}} v}\ \textsc{MatchRaise}$$

$$\frac{E \vdash t \Downarrow_m v \qquad E, x_m \mapsto v \vdash u_m \Downarrow_{m'} v'}{E \vdash \mathsf{dispatch}\ t\ \mathsf{with\ val}\ x_{\mathsf{val}} \Rightarrow u_{\mathsf{val}} \mid \mathsf{exn}\ x_{\mathsf{exn}} \Rightarrow u_{\mathsf{exn}} \Downarrow_{m'} v'}\ \textsc{Dispatch}$$

Fig. 1. Big-step semantics.

The non-ambiguous pattern-matching simplifies the semantics of the term match t with $p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$, as only one pattern can match the value of t, and thus only one branch is considered during the evaluation.

The rule Dispatch deals with exception handling: the evaluation of the term dispatch t with val $x_{\mathsf{val}} \Rightarrow u_{\mathsf{val}} \mid$ exn $x_{\mathsf{exn}} \Rightarrow u_{\mathsf{exn}}$ first evaluates t. If t reduces to a value, then the value branch u_{val} is evaluated. Otherwise, if t raises an exception, the exception branch u_{exn} is evaluated. In both cases, the value or the exception is added to the environment of the corresponding branch.

4 An Abstract Domain for Regular Sets of Values

In this section, we define an abstract domain that is able to represent inductively defined sets of values of our programming language. It is parameterised over a non-relational, numeric abstract domain $\mathbb{I}$, that provides a concretisation function $\gamma_{\mathbb{I}} : \mathbb{I} \to \wp(\mathbb{Z})$, a test for the abstract inclusion pre-order, and operations for union, intersection and widening, with the standard soundness conditions. For instance, the soundness of abstract union is stated: $\gamma_{\mathbb{I}}(\mathsf{I}_1) \cup \gamma_{\mathbb{I}}(\mathsf{I}_2) \subseteq \gamma_{\mathbb{I}}(\mathsf{I}_1 \sqcup_{\mathbb{I}} \mathsf{I}_2)$.

The definition of our abstract domain follows:

Definition 2 (Abstract values).

$$\begin{array}{rcll}
\mathsf{A} \in \mathbb{A} & ::= & \{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\} \mid \alpha \mid \mu\alpha.\mathsf{A} & \textit{(Abstract value)} \\
\mathsf{I} \in \mathbb{I} & ::= & \textit{any numeric abstract domain} & \textit{(Abstract integers)} \\
\mathsf{C} & ::= & \{\overline{c \mapsto (\mathsf{A}, \dots, \mathsf{A})}\} \mid \top & \textit{(Abstract constructs)} \\
\mathsf{F} & ::= & \{\overline{\mu f.\,\lambda x.\,t \mapsto \mathsf{E}}\} \mid \top & \textit{(Abstract closures)} \\
\mathsf{E} \in \mathbb{E} & ::= & \{\overline{x \mapsto \mathsf{A}}\} & \textit{(Abstract environment)}
\end{array}$$

An abstract value, written A, describes which integers it denotes (in the field ints), *and* which values whose head is a constructor it denotes (in the field $\mathsf{constructs}$), *and* which function closures it denotes (in the field funs). The integer values are described by a numeric abstract domain that is taken as parameter.

The constructed values are described by a map whose keys are the possible head constructors of the values, and whose data are tuples of abstract values, that denote the possible values for all the arguments of that constructor. The constructed values might also be described by $\top$, which means that the head constructor could be any constructor, and the arguments may be any value.

Similarly, the possible function closures are described by a map that associates possible codes of the function to abstract environments. The environments map free variables of the corresponding function code to abstract values, denoting the possible concrete values of these variables. The closures might also be described by $\top$, to represent any closure made from any function code with any environment.

Finally, we can construct recursive sets of values through the use of variables α, that are introduced by the μ constructor of fixpoints.

The bottom value is $\{\mathsf{ints} : \bot;\ \mathsf{constructs} : \{\};\ \mathsf{funs} : \{\}\}$, and the top value is $\{\mathsf{ints} : \top;\ \mathsf{constructs} : \top;\ \mathsf{funs} : \top\}$. We may completely omit some of the fields (ints, $\mathsf{constructs}$ or funs) when they are associated with a bottom value.

This informal explanation is formalised in the concretisation function:

Definition 3 (Concretisation). *Assume Γ is a finite mapping from variables to abstract values. The concretisation $\gamma_\Gamma : \mathbb{A} \to \wp(\mathbb{V})$ is defined by $\gamma_\Gamma \{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\} = \gamma(\mathsf{I}) \cup \gamma_\Gamma(\mathsf{C}) \cup \gamma_\Gamma(\mathsf{F})$, where:*

$$\begin{aligned}
\gamma_\Gamma(\alpha) &= \Gamma(\alpha)\\
\gamma_\Gamma(\mu\alpha.\mathsf{A}) &= \mathrm{lfp}_{\subseteq}(\lambda S.\gamma_{\Gamma,\alpha:S}(\mathsf{A}))\\
\gamma(\mathsf{I}) &= \gamma_{\mathbb{I}}(\mathsf{I})\\
\gamma_\Gamma(\mathsf{C}) &= \begin{cases} \{c(v_1,\dots,v_n) \mid c \in \mathcal{C} \wedge \forall 1 \le i \le n, v_i \in \mathbb{V}\} & \textit{if } \mathsf{C} = \top\\ \left\{ c(v_1,\dots,v_n) \,\middle|\, \begin{array}{l} (c \mapsto (\mathsf{A}_1,\dots,\mathsf{A}_n)) \in \mathsf{C}\\ \wedge \forall 1 \le i \le n, v_i \in \gamma_\Gamma(\mathsf{A}_i) \end{array} \right\} & \textit{otherwise} \end{cases}\\
\gamma_\Gamma(\mathsf{F}) &= \begin{cases} \{\langle E, \mu f.\, \lambda x.\, t\rangle \mid E \in \mathbb{E} \wedge t \in \mathbb{T}\} & \textit{if } \mathsf{F} = \top\\ \{\langle E, \mu f.\, \lambda x.\, t\rangle \mid (\mu f.\, \lambda x.\, t \mapsto \mathsf{E}) \in \mathsf{F} \wedge E \in \gamma_\Gamma(\mathsf{E})\} & \textit{otherwise} \end{cases}\\
\gamma_\Gamma(\mathsf{E}) &= \{E \mid \mathrm{dom}\, E = \mathrm{dom}\, \mathsf{E} \wedge \forall x \in \mathrm{dom}\, E, E(x) \in \gamma_\Gamma(\mathsf{E}(x))\}
\end{aligned}$$

The definition is justified by the fact that the function $\lambda S.\gamma_{\Gamma,\alpha:S}(\mathsf{A})$ is monotonic, and thus has a least fixed point, thanks to the Knaster-Tarski theorem. This is formalised by the following lemma:

Lemma 1. *Consider the inclusion order $\subseteq$ on $\wp(S)$, and its pointwise extension on environments Γ. For any abstract value A, the function $\lambda\Gamma.\gamma_\Gamma(\mathsf{A})$ is monotonic.*

The fact that our abstract values may represent sets of values that might not all have the same types may seem surprising, since our goal is, ultimately, to analyse strongly typed programs. The crux of the explanation lies in the fact that our abstract domain can only represent *regular* sets of values. If we restricted our abstract values so that they represent homogeneously typed values, it would be difficult to represent sets of values that are induced by a non-regular recursive type—like the type of finger trees [23]—or by generalised algebraic data types (GADTs). Indeed, one would need to find an over-approximation of such sets, and we would often approximate with the $\top$ abstract value. The ability to describe regular sets of values that may not have all the same type gives us more freedom, and allows to find more precise approximations. For instance, we can represent finger trees as a recursive set whose values are either trees or fingers, although trees and fingers have distinct types. In practice, the $\top$ value is never produced.

We write $\mathsf{A}_1[\alpha \leftarrow \mathsf{A}_2]$ to denote the capture avoiding substitution. We write $\gamma(\mathsf{A})$ for $\gamma_{[]}(\mathsf{A})$, *i.e.*, when the environment is empty.

The unwinding of fixpoints preserves the concretisation of abstract values.

Lemma 2 (Unwinding). $\gamma(\mu\alpha.\mathsf{A}) = \gamma(\mathsf{A}[\alpha \leftarrow \mu\alpha.\mathsf{A}])$

To define several operations on abstract values, we restrict them to *well-formed* values, using the standard contractiveness property for recursive types [16]:

Definition 4 (Contractiveness). *An abstract value* $\mathsf{A} = \mu\beta_1.\ldots\mu\beta_n.\mathsf{A}'$ *is* α*-contractive if* $n \geq 0$ *and* A' *does not start with* μ *and is not the variable* α*.*

Well-formedness requires that fixpoints must be contractive, that constructors are used with the correct arity, and that the environment in closures only define bindings for the free variables of the functions.

Definition 5 (Well-formedness). *An abstract value* A *is* well-formed *when the following conditions are satisfied:*

- *For any* $\mu\alpha.\mathsf{A}'$ *that occurs in* A*, the abstract value* A' *is* α*-contractive, and*
- *For any* $c \mapsto (\mathsf{A}_1, \ldots, \mathsf{A}_n)$ *that occurs in* A*, the arity of* c *is* n*, and*
- *For any* $\mu f.\, \lambda x.\, t \mapsto \mathsf{E}$ *that occurs in* A*,* $\operatorname{dom} \mathsf{E} = \operatorname{fv}(\mu f.\, \lambda x.\, t)$*.*

Well-formedness rules out the abstract value $\mu\alpha.\alpha$, whose concretisation is the empty set. Well-formedness is preserved by substitution, provided contractiveness for the substituted variable is satisfied. This ensures that unwinding fixpoints preserves well-formedness. In the rest of this article, we only consider closed, well-formed abstract values.

For any abstract value A, we can retrieve the subset of integer values (respectively, constructed values, or function closures) by unwinding the top-level μs if there are any, and eventually getting the ints field (respectively, $\mathsf{constructs}$, or funs). This is formalised in the following definition for projection on integers:

Definition 6 (Projection on integers). *The projection on integers of a well-formed abstract value* A*, written* $\mathsf{A.ints}$*, is defined as follows:*

$$\{\mathsf{ints} : \mathsf{I};\ \mathsf{constructs} : \mathsf{C};\ \mathsf{funs} : \mathsf{F}\}.\mathsf{ints} = \mathsf{I}$$
$$(\mu\alpha.\mathsf{A}).\mathsf{ints} = (\mathsf{A}[\alpha \leftarrow \mu\alpha.\mathsf{A}]).\mathsf{ints}$$

The definition for projection is well founded, thanks to the contractiveness of μs: only a finite number of unwindings is necessary. The projections $\mathsf{A.constructs}$ and $\mathsf{A.funs}$ are defined in a similar way. Projection on integers is sound, as it over-approximates the set of integers an abstract value contains:

Lemma 3 (Soundness of projection on integers). $\gamma(\mathsf{A}) \cap \mathbb{Z} \subseteq \gamma(\mathsf{A.ints})$

Projections for constructors and closures enjoy similar soundness properties.

4.1 Inclusion, Union and Intersection

Following the methodology employed in the context of recursive subtyping, we define the inclusion relation between abstract values as a *co-inductive* relation.

Definition 7 (Abstract inclusion). *The inclusion between abstract values, written* $\mathsf{A}_1 \sqsubseteq \mathsf{A}_2$ *is defined as a co-inductive relation by the following rules:*

$$\frac{\mathsf{A}_1[\alpha \leftarrow \mu\alpha.\mathsf{A}_1] \sqsubseteq \mathsf{A}_2}{\mu\alpha.\mathsf{A}_1 \sqsubseteq \mathsf{A}_2} \qquad \frac{\mathsf{A}_1 \neq \mu\beta.\mathsf{A}_1' \qquad \mathsf{A}_1 \sqsubseteq \mathsf{A}_2[\alpha \leftarrow \mu\alpha.\mathsf{A}_2]}{\mathsf{A}_1 \sqsubseteq \mu\alpha.\mathsf{A}_2}$$

$$\frac{\mathsf{I}_1 \sqsubseteq_\mathsf{I} \mathsf{I}_2 \qquad \mathsf{C}_1 \sqsubseteq_\mathsf{C} \mathsf{C}_2 \qquad \mathsf{F}_1 \sqsubseteq_\mathsf{F} \mathsf{F}_2}{\{\mathsf{ints}:\mathsf{I}_1;\ \mathsf{constructs}:\mathsf{C}_1;\ \mathsf{funs}:\mathsf{F}_1\} \sqsubseteq \{\mathsf{ints}:\mathsf{I}_2;\ \mathsf{constructs}:\mathsf{C}_2;\ \mathsf{funs}:\mathsf{F}_2\}}$$

$$\frac{}{\mathsf{C}_1 \sqsubseteq_\mathsf{C} \top} \qquad \frac{}{\mathsf{F}_1 \sqsubseteq_\mathsf{F} \top}$$

$$\frac{\begin{array}{c}\forall (c \mapsto (\mathsf{A}_{1,1}, \dots, \mathsf{A}_{1,n})) \in \mathsf{C}_1,\\ \exists (c \mapsto (\mathsf{A}_{2,1}, \dots, \mathsf{A}_{2,n})) \in \mathsf{C}_2,\\ \forall 1 \leq i \leq n,\ \mathsf{A}_{1,i} \sqsubseteq \mathsf{A}_{2,i}\end{array}}{\mathsf{C}_1 \sqsubseteq_\mathsf{C} \mathsf{C}_2} \qquad \frac{\begin{array}{c}\forall (\mu f.\,\lambda x.\,t \mapsto \mathsf{E}_1) \in \mathsf{F}_1,\\ \exists (\mu f.\,\lambda x.\,t \mapsto \mathsf{E}_2) \in \mathsf{F}_2,\\ \forall x \in \operatorname{dom} \mathsf{E}_1,\ \mathsf{E}_1(x) \sqsubseteq \mathsf{E}_2(x)\end{array}}{\mathsf{F}_1 \sqsubseteq_\mathsf{F} \mathsf{F}_2}$$

In this definition, the relation $\sqsubseteq_\mathsf{I}$ is provided by the abstract domain on integers. The inclusion relation unfolds fixpoints when necessary, and otherwise compares each field (integers, constructed values, closures) separately, by treating the finite maps for constructed values and closures as *disjunctions*, *i.e.*, by using the standard Hoare ordering. In practice, the inclusion test is implemented by transforming abstract values into graphs that resemble tree automata: each graph node corresponds to a sub-term of an abstract value, and μ-nodes create cycles. Then, it suffices to check whether one automaton simulates the other [1,31,16].

Lemma 4 (Inclusion is a pre-order). *The inclusion between closed, well-formed abstract values is a pre-order, i.e., a reflexive and transitive relation.*

The definitions for abstract union and intersection are defined in the companion research report [34] in a similar way, as co-inductive relations that unwind fixpoints when needed.

The abstract operations enjoy the expected soundness properties:

Lemma 5 (Soundness of abstract operations). *For any closed, well-formed abstract values* A_1 *and* A_2*:*

- $\mathsf{A}_1 \sqsubseteq \mathsf{A}_2$ *implies* $\gamma(\mathsf{A}_1) \subseteq \gamma(\mathsf{A}_2)$, *and*
- $\gamma(\mathsf{A}_1) \cup \gamma(\mathsf{A}_2) \subseteq \gamma(\mathsf{A}_1 \sqcup \mathsf{A}_2)$, *and*
- $\gamma(\mathsf{A}_1) \cap \gamma(\mathsf{A}_2) \subseteq \gamma(\mathsf{A}_1 \sqcap \mathsf{A}_2)$.

The proof of Lemma 5 crucially relies on Lemma 2, that proves that unwinding a recursive value preserves its concretisation.

Union and intersection are implemented by translating the values into graphs, on which union and intersection are easily computed. Then, we transform them back into trees with μ nodes. Our implementation exploits the *locally nameless* representation [5], where bound variables are encoded as de Bruijn indices. We leverage this canonical representation by *hash-consing* values and *memoising* the operations [13]. This has proved essential to obtain acceptable performance.

4.2 Widening

The widening, written $\mathsf{A}_1 \nabla \mathsf{A}_2$, is a binary operator on abstract values that over-approximates the union of abstract values, and is used to approximate the Kleene fixpoint iterations. The role of the widening is central in abstract interpretation, as it serves two purposes. Firstly, the widening must find generalisations of abstract values, in order to find invariants. This part impacts the *precision* of the analysis, and relies on heuristics. Secondly, it must ensure the *termination* of the analysis, by enforcing a stability property: every widening chain must reach a limit in finite time. This part impacts the *performance* of the analyser.

In our abstract domain, the widening operator is responsible for finding *regularities* in abstract values and for creating μ nodes. A similar idea was used in the analysis of Prolog programs using *type graphs* [22], that are trees that contain cycles. Our widening draws inspiration from type graphs.

We now give the informal procedure to compute the widening of two abstract values A_1 and A_2. It operates in two phases. The first phase proceeds as follows:

1. Compute the union A_{12} of A_1 and A_2 where the widening of the numeric abstract domain is used, instead of the standard union. This ensures that the numeric parts of abstract values won't grow indefinitely.
2. Compute $\mathsf{A}_{\mathsf{new}}$, which is a *minimised* version of A_{12}. Minimisation is performed by an algorithm on tree automata, that produces a semantically equivalent abstract value, and whose size is smaller.
3. Compare the $\mathsf{A}_{\mathsf{new}}$ and A_1 (viewed as trees):
   - If the height of $\mathsf{A}_{\mathsf{new}}$ is not greater than the height of A_1, return $\mathsf{A}_{\mathsf{new}}$;
   - If, for each construct and each code of closures, the maximal number of occurrences in each tree path of $\mathsf{A}_{\mathsf{new}}$ is less than those occurrences in A_1, or a user-provided threshold, return $\mathsf{A}_{\mathsf{new}}$;
   - Otherwise, go to the *shrinking phase.*

Steps 2 and 3 allow the size of abstract values to grow enough, before a shrinking phase starts. In practice, this is important to find precise invariants.

The *shrinking phase*, which takes inspiration from the widening operation of *type graphs*, tries to *shrink* $\mathsf{A}_{\mathsf{new}}$, by introducing μ nodes at appropriate positions to "fold the abstract value on itself". It proceeds as follows:

1. Find *clashes* between A_1 and $\mathsf{A}_{\mathsf{new}}$, *i.e.*, nodes that are reachable through the same path (possibly unwinding μ nodes) in the two trees, and such that:
   - Either, the two nodes have different sets of head constructors or codes of functions: this means that the two nodes might differ *semantically.*

  - Or, the two nodes have different *depths* in the two trees: this means that some path was followed through a μ-unwinding.
2. If no clash is found, then return $\mathsf{A}_{\mathsf{new}}$.
3. If a clash is found, then we try to create a cycle in $\mathsf{A}_{\mathsf{new}}$ by merging the clashing node with one of its ancestors:
   - We search for the closest ancestor of the clashing node that is *semantically larger* in the sense of the pre-order. If there is such an ancestor, then we merge it with the clashing node, thus creating a cycle.
   - If no such ancestor exists, we search for the closest ancestor that has *at least the same head constructors and function codes* as the clashing node, then we merge it with the clashing node too.
   - If no such ancestor exists, then we return $\mathsf{A}_{\mathsf{new}}$ unchanged, which allows the abstract values to grow.

   We repeat this operation until no clashing node remains, or until a maximal number of iterations is reached. In the latter case, we *truncate* $\mathsf{A}_{\mathsf{new}}$, *i.e.*, we replace some nodes with $\top$, so that it has the same height as A_1.

In practice, we could not find any case where the final truncation is needed. We have observed that our widening operator finds precise generalisations in practice.

5 An Abstract Interpreter to Detect Uncaught Exceptions

To design our abstract interpreter, we took inspiration from the "Abstracting Definitional Interpreter" approach [11]. This methodology prescribes to derive an abstract interpreter from a concrete big-step interpreter that computes in a monad, that is a parameter of the interpreter. Furthermore, the methodology fosters the use of the *open recursive style*: the interpreter should be a function that takes as extra parameter the function that was intended to be called recursively.

The first aspect—being parameterised by a monad—is motivated by the fact that one could use a monad that computes over *abstractions* of values. In §5.1, we present a monad that is an abstraction of the exception monad. It is also an abstract domain, and is therefore well suited to define an abstract interpreter.

The second aspect—using open recursive style—permits the use of dynamic fixpoint solvers [59,63,12,24,6,30]. Such solvers compute post-fixpoints, *i.e.*, over-approximations of solutions of systems of equations over abstract values, for which the set of equations might be discovered dynamically, while solving the equations. New equations can be discovered, for instance, when the control flow of a program depends on its data flow. This is the case of higher-order programs, as the function that can be called at a given call site can possibly result from a computation. We present in §5.2 our abstract interpreter as a function that computes in the abstract exception monad, and is defined in open recursive style.

5.1 The Abstract Exception Monad

A big-step interpreter for a programming language with exceptions can be defined in an elegant manner using the *exception monad*, which we briefly recall. In the exception monad, a computation is either a success value, or an exception that carries some value—typically of type exception—from the object language.

$$\mathsf{type\ m}\,\beta = \mathsf{Success}\,\beta \mid \mathsf{Exception}\,\mathbb{V}$$

$$\begin{array}{ll} \mathbf{return} :: \beta \rightarrow \mathsf{m}\,\beta & \gg\!= \;::\; \mathsf{m}\,\beta_1 \rightarrow (\beta_1 \rightarrow \mathsf{m}\,\beta_2) \rightarrow \mathsf{m}\,\beta_2 \\ \mathbf{return}\,x = \mathsf{Success}\,x & (\mathsf{Success}\,x) \gg\!= f = f\,x \\ & (\mathsf{Exception}\,e) \gg\!= f = \mathsf{Exception}\,e \end{array}$$

In this monad, the **raise** function expresses the action of throwing an exception, while the **dispatch** function, corresponds to the dispatch construct of our prototype language (§3), and expresses the action of catching an exception.

$$\begin{array}{ll} \mathbf{raise} :: \mathbb{V} \rightarrow \mathsf{m}\,\mathbb{A} & \mathbf{dispatch} :: \mathsf{m}\,\beta_1 \rightarrow (\beta_1 \rightarrow \mathsf{m}\,\beta_2) \rightarrow (\mathbb{V} \rightarrow \mathsf{m}\,\beta_2) \rightarrow \mathsf{m}\,\beta_2 \\ \mathbf{raise}\,e = \mathsf{Exception}\,e & \mathbf{dispatch}\,(\mathsf{Success}\,x)\,f\,g = f\,x \\ & \mathbf{dispatch}\,(\mathsf{Exception}\,e)\,f\,g = g\,e \end{array}$$

The **raise** function simply injects its argument into the exception case, whereas the **dispatch** function takes two continuations, to handle, respectively, the success case, and the exception case, by performing a case analysis on the monadic value.

We can easily define a monad that mimics the behaviour of the exception monad, with the difference that it deals with abstractions of sets of (possibly exceptional) values, instead of mere exceptional values. The construction is based on the observation that $\wp(\mathsf{m}\,\beta)$ is isomorphic to $\wp(\beta) \times \wp(\mathbb{V})$, that can itself be abstracted into $\wp(\beta) \times \wp(\mathbb{A})$ by using our abstract domain for sets of values. Thus, we define the abstract exception monad, written $\mathsf{m}^\sharp\,\beta$, as follows:

$$\mathsf{type\ m}^\sharp\,\beta = \beta \times \mathbb{A}$$

$$\begin{array}{ll} \mathbf{return}^\sharp :: \beta \rightarrow \mathsf{m}^\sharp\,\beta & \gg\!=^\sharp \;::\; \mathsf{m}^\sharp\,\beta_1 \rightarrow (\beta_1 \rightarrow \mathsf{m}^\sharp\,\beta_2) \rightarrow \mathsf{m}^\sharp\,\beta_2 \\ \mathbf{return}^\sharp B = (B, \bot) & (B, \mathsf{A}) \gg\!=^\sharp f = \text{let } (B', \mathsf{A}') = f\,B \text{ in } (B', \mathsf{A} \sqcup \mathsf{A}') \end{array}$$

The **return**$^\sharp$ operation records its argument as the set of possible values, and asserts that no exception is returned: the set of possible exceptions is $\bot$. The $\gg\!=^\sharp$ operation retrieves the value part of its monadic argument and passes it to the continuation. The final value is composed of the value part that was produced by the continuation, and of the union of the exceptions that might have been raised by the monadic value and by the evaluation of the continuation. The functions **return**$^\sharp$ and $\gg\!=^\sharp$ satisfy the *monad laws* if $(\bot, \sqcup)$ is a monoid.

The fact that $\mathsf{m}^\sharp\,\beta$ is a monad does not suffice to use it in an abstract interpreter, though. We also need $\mathsf{m}^\sharp\,\beta$ to be an abstract domain, *i.e.*, one must decide when two monadic values are included in each other, and how to compute abstract unions, intersections, and widening.

Interestingly, the monad $\mathsf{m}^\sharp\,\beta$ acts as an abstract domain as soon as β is an abstract domain: this is the standard cartesian product of abstract domains, where operations are defined pointwise. In practice, we only need to consider the instance $\mathsf{m}^\sharp\,\mathbb{A}$, *i.e.*, the domain of exceptional abstract values.

The remaining pieces that are needed to use $\mathsf{m}^\sharp\,\beta$ in an abstract interpreter are the abstract versions of **raise** and **dispatch**. They are defined as follows:

$$\begin{array}{ll}\textbf{raise}^\sharp :: \mathbb{A} \to \mathsf{m}^\sharp\,\mathbb{A} & \textbf{dispatch}^\sharp\ : \mathsf{m}^\sharp\,\beta \to (\beta \to \mathsf{m}^\sharp\,\mathbb{A}) \to (\mathbb{A} \to \mathsf{m}^\sharp\,\mathbb{A}) \to \mathsf{m}^\sharp\,\mathbb{A}\\ \textbf{raise}^\sharp \mathsf{A} = (\bot, \mathsf{A}) & \textbf{dispatch}^\sharp\ (B, \mathsf{A})\ F\ G = F\,B \sqcup G\,\mathsf{A}\end{array}$$

The **raise**$^\sharp$ operation raises a set of possible exceptions, by recording the abstract value for exceptions in the set of possibly returned exceptions, and by returning the bottom value, since it can never return any value. It is the dual of **return**$^\sharp$.

The **dispatch**$^\sharp$ function executes the value continuation on the set of possible values, and executes the exception continuation on the set of possible exceptions, and then returns their abstract union in the domain of exceptional values.

We can easily show that the abstract operations compute over-approximations of their counterpart in the exception monad. Assume the type β is equipped with a concretisation function $\gamma_\beta : \beta \to \wp(\mathbb{B})$ for some set $\mathbb{B}$. Then, we define the concretisation for the abstract monad:

$$\begin{array}{l}\gamma_{\mathsf{m}^\sharp\,\beta} : \mathsf{m}^\sharp\,\beta \to \wp(\mathsf{m}\,\mathbb{B})\\ \gamma_{\mathsf{m}^\sharp\,\beta}(B, \mathsf{A}) = \{\mathsf{Success}\,b \mid b \in \gamma_\beta(B)\} \cup \{\mathsf{Exception}\,v \mid v \in \gamma(\mathsf{A})\}\end{array}$$

The concretisation specifies that the first component of monadic values form the success values, and that the second component describe possible exceptions.

The soundness results for the abstract operations show that they compute over-approximations of their concrete counterparts:

Lemma 6. *The following inclusions are satisfied:*

- $\{\textbf{return}b \mid b \in \gamma_\beta(B)\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta}(\textbf{return}^\sharp B)$
- $\{m \ggg= f \mid m \in \gamma_{\mathsf{m}^\sharp\,\beta_1}(M), f \in \gamma_{\beta_1 \to \mathsf{m}^\sharp\,\beta_2}(F)\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta_2}(M \ggg=^\sharp F)$
- $\{\textbf{raise}v \mid v \in \gamma(\mathsf{A})\} \subseteq \gamma_{\mathsf{m}^\sharp\,\mathbb{A}}(\textbf{raise}^\sharp \mathsf{A})$
- $\left\{ \textbf{dispatch}\,m\,f\,g \;\middle|\; \begin{array}{l} m \in \gamma_{\mathsf{m}^\sharp\,\beta_1}(M),\\ f \in \gamma_{\beta_1 \to \mathsf{m}^\sharp\,\beta_2}(F),\\ g \in \gamma_{\mathbb{V} \to \mathsf{m}^\sharp\,\beta_2}(G) \end{array} \right\} \subseteq \gamma_{\mathsf{m}^\sharp\,\beta_2}(\textbf{dispatch}^\sharp M\,F\,G)$

where $\gamma_{\beta_1 \to \beta_2}(F) = \{f \mid \forall X, \forall x \in \gamma_{\beta_1}(X), f\,x \in \gamma_{\beta_2}(F\,X)\}$.

5.2 A Monadic Abstract Interpreter in Open Recursive Style

In this section, we describe our whole-program static analyser. It infers an over-approximation of the values that a program might compute, and the exceptions that it might raise, with the possible values they carry. Although it analyses programs that can deal with first-class functions, it is *not* defined as a control-flow analyser [60], but rather as an abstract interpreter that performs a value analysis. The insight is the following: since functions are first-class citizens in the language, a value analysis also infers an approximation of the control flow. A value analysis will indeed compute which functions may be called at every call site.

Our analyser follows the *open recursive style*, and has the following type:

$$(\mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}) \to (\mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A})$$

$$
\begin{aligned}
&\text{Assuming } \mathsf{eval} :: \mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}, \quad \text{we define } [\![\cdot]\!]^{\mathsf{eval}}_{\cdot} :: \mathbb{T} \to \mathbb{E} \to \mathsf{m}^\sharp\,\mathbb{A}\\
[\![x]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp\,\mathsf{E}(x)\\
[\![c(x_1,\ldots,x_n)]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \begin{cases}\mathbf{return}^\sharp\bot \text{ if } \mathsf{E}(x_i) = \bot \text{ for some } 1 \le i \le n\\ \mathbf{return}^\sharp\{\mathsf{constructs} : \{c \mapsto (\mathsf{E}(x_1),\ldots,\mathsf{E}(x_n))\}\}\\ \qquad \text{otherwise}\end{cases}\\
[\![n]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp\{\mathsf{ints} : \{n\}\}\\
[\![x_1\ op\ x_2]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp\{\mathsf{ints} : \mathsf{E}(x_1).\mathsf{ints}\ [\![op]\!]\ \mathsf{E}(x_2).\mathsf{ints}\}\\
[\![\mu f.\,\lambda x.\,t]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{return}^\sharp\{\mathsf{funs} : \{\mu f.\,\lambda x.\,t \mapsto \mathsf{E}|_{\mathrm{fv}(\mu f.\,\lambda x.\,t)}\}\}\\
[\![x\ y]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \text{if } \mathsf{E}(y) = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else}\\
&\quad \textstyle\bigsqcup_{(\mu f.\,\lambda x.t \mapsto \mathsf{E}') \in \mathsf{E}(x).\mathsf{funs}} \mathsf{eval}\ t\ \mathsf{E}''\\
&\quad \text{where } \mathsf{E}'' = \mathsf{E}', f \mapsto F, x \mapsto \mathsf{E}(y)\\
&\quad \text{and } F = \{\mathsf{funs} : \{\mu f.\,\lambda x.t \mapsto \mathsf{E}'\}\}\\
[\![\mathsf{let}\ x = t\ \mathsf{in}\ u]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \ggg^\sharp \lambda v.\,\text{if } v = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else}\\
&\quad [\![u]\!]^{\mathsf{eval}}_{\mathsf{E},x:v}\\
[\![\mathsf{match}\,t\,\mathsf{with}\,p_1 \Rightarrow t_1 \mid \cdots \mid p_n \Rightarrow t_n]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \ggg^\sharp \lambda v.\text{if } v = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else}\\
&\quad \textstyle\bigsqcup_{1 \le i \le n} (p_i \prec\!\!\prec^\sharp v) \ggg^\sharp \lambda\mathsf{E}'.\\
&\qquad \text{if } \mathsf{E}' = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else } [\![t_i]\!]^{\mathsf{eval}}_{\mathsf{E},\mathsf{E}'}\\
[\![\mathsf{raise}\ x]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{raise}^\sharp\,\mathsf{E}(x)\\
[\![\mathsf{dispatch}\,u\,\mathsf{with\,val}\ x \Rightarrow t \mid \mathsf{exn}\ y \Rightarrow r]\!]^{\mathsf{eval}}_{\mathsf{E}} &= \mathbf{dispatch}^\sharp\ [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}\\
&\quad (\lambda v.\,\text{if } v = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else } [\![u]\!]^{\mathsf{eval}}_{\mathsf{E},x:v})\\
&\quad (\lambda e.\,\text{if } e = \bot \text{ then } \mathbf{return}^\sharp\bot \text{ else } [\![r]\!]^{\mathsf{eval}}_{\mathsf{E},y:e})
\end{aligned}
$$

Fig. 2. Definition of the abstract interpreter.

It takes as a parameter an analyser, that represents the information that has been discovered so far on the program, and produces an analyser as output, that exploits the input analyser to produce more analysis results, that are possibly less precise. The role of the fixpoint solver is to find a post-fixpoint of this functional. Similar approaches—leveraging fixpoint solvers to define static analysers—have been successfully used in other work on static analysis [22,64,50,4].

Our abstract interpreter is defined in Figure 2, where $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}$ denotes the abstract value of type $\mathsf{m}^\sharp\,\mathbb{A}$ obtained by analysing the program t under the abstract environment E, and using the analysis function eval for recursive calls. Importantly, the analyser does not call eval for *every* recursive call. Instead, eval is only used when the analyser cannot be called on a strict sub-term. In practice, this means that eval is only used to analyse function calls. In every other place, we have the guarantee that the analysis is demanded on a strict sub-term, and a standard recursive call is performed. This strategy saves time in practice, as it lightens the burden of the fixpoint solver, that only needs to find post-fixpoints for function calls rather than for every program point.

To analyse a variable, we return the abstract value found in the environment.

To analyse a construct, we retrieve the abstract values for every argument, and return the corresponding abstract value for that constructor, or $\bot$ if some of the argument was $\bot$, because of the *eager* semantics.

The analysis of an integer returns this integer injected in the integer domain. The analysis of binary operations on integers retrieves the integer parts of the abstract values for the two arguments, and returns the result of the transfer function from the integer domain for that binary operation.

The analysis of a function mimics the concrete semantics: it returns an abstract closure composed of the code of the function and its abstract environment.

The analysis of function calls is more interesting. If the abstract value for the argument is $\bot$, then we return $\bot$, because evaluation is eager. Otherwise, we retrieve all the possible closures for the value at the call position, and analyse their bodies by extending their environments with the abstract value for the argument, and with the abstract closure itself (we are dealing with recursive functions). The final result is the union—at the level of the abstract monad—of the analyses of all the possible function bodies. Because the bodies of the functions that are analysed are not strict sub-terms of the original term $x\,y$, we perform an external recursive call to the analyser, by using the eval parameter.

The analysis of let bindings chains the analyses of its two parts, and, because evaluation is eager, checks for emptiness before analysing the second sub-term.

The pattern matching construct is analysed by first analysing the scrutinee, and then analysing each branch of the match *independently*. For each branch, we retrieve the environment produced by matching the abstract value with the pattern (written $p \prec\!\!\prec^\sharp v$), and then we analyse the code of that branch if the matching was possible. Then, we take the union—at the level of the abstract monad—of the analysis results from each branch. Notably, the exceptions that any branch might raise are reported in the final result. The definition for matching abstract values against patterns is available in the companion research report [34].

Analysing the raise construct is easy: a call to the $\mathbf{raise}^\sharp$ function suffices. Finally, the analysis of $\mathsf{dispatch}$ amounts to calling the $\mathbf{dispatch}^\sharp$ function from the abstract monad, on the analysis of the scrutinee, and on two continuations, that will analyse the codes of the two branches, if they are given non-$\bot$ arguments.

5.3 Soundness of the Abstract Interpreter

We show that the abstract interpreter of Figure 2 is *sound*, in the sense that it computes an over-approximation of the behaviour of programs.

Definition 8 (Behaviour of programs). *Let S be a set of evaluation environments:* $\mathrm{EVAL}_S\, t = \bigcup_{E \in S} \{\mathsf{Success}\, v \mid E \vdash t \Downarrow_{\mathsf{val}} v\} \cup \{\mathsf{Exception}\, e \mid E \vdash t \Downarrow_{\mathsf{exn}} e\}$

The behaviour of a program t as a function EVAL that takes a set of evaluation environments as input, and produces a set of values with a tag that indicates whether it results from normal or from exceptional evaluation.

Then, the soundness of the abstract interpreter follows:

Theorem 1 (Soundness). *Assume* eval *is a post-fixpoint, i.e.,* $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \sqsubseteq \mathsf{eval}\, t\, \mathsf{E}$ *for every t and* E*. Then,* $\mathrm{EVAL}_{\gamma(\mathsf{E})}\, t \subseteq \gamma_{\mathsf{m}\,\mathbb{A}}([\![t]\!]^{\mathsf{eval}}_{\mathsf{E}})$.

Proof. We have to show that for every $E \in \gamma(\mathsf{E})$, $m \in \{\mathsf{val}, \mathsf{exn}\}$ and $v \in \mathbb{V}$, if $E \vdash t \Downarrow_m v$, then $r \in \gamma_{\mathsf{m}\,\mathbb{A}}([\![t]\!]^{\mathsf{eval}}_{\mathsf{E}})$, where $r = \mathsf{Success}\, v$ when $m = \mathsf{val}$, and $r = \mathsf{Exception}\, v$ when $m = \mathsf{exn}$. The proof proceeds by induction on the evaluation judgement, generalising over m and E. The only interesting case is the one for function application, which exploits the induction hypothesis, the post-fixpoint property of eval and the soundness of abstract inclusion $\sqsubseteq$. All other cases result from the soundness of the abstract operations and from induction hypotheses. □

The soundness theorem assumes that eval is a post-fixpoint, *i.e.*, $[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}} \sqsubseteq \mathsf{eval}\, t\, \mathsf{E}$. This property is ensured by the soundness of the fixpoint solver, that always returns a post-fixpoint. The function eval is, indeed, the result of the fixpoint solver called on the function $\lambda \mathsf{eval}.\lambda t.\lambda \mathsf{E}.[\![t]\!]^{\mathsf{eval}}_{\mathsf{E}}$.

6 An Abstract Interpreter for OCaml Programs

Based on the abstract interpreter of §5, we implemented a static analyser for OCaml programs (version 4.14), that returns a map from top-level identifiers of the program to their abstract values. Our prototype and its test suite (see §7) are available as a companion artefact [35].

We have implemented several optimisations, that are crucial to obtain decent performance. For example, nodes of the analysed AST are indexed by program points using unique integers as identifiers. This enables efficient comparison of sub-terms and allows using efficient data structures like Patricia trees [53]. Moreover—this is of paramount importance for performance—we perform *hash-consing* of abstract values and *memoise* the operations on these abstract values.

We present in the next sections some key implementation details that we needed to analyse OCaml programs.

6.1 Refinements With Respect to the Formal Presentation

The abstract interpreter we implemented follows the structure we have presented in §5.2, but implements three more refinements, that we purposely elided to follow the presentation more easily. A thorough presentation of these refinements would go beyond the scope of the current paper.

Context sensitivity. Our analyser is *context sensitive*: we implemented a form of *call site* sensitivity, that is akin to an abstraction of the call stack. Following [50], we retain full sensitivity until the list of call sites becomes *maximal*, *i.e.*, when a program point appears more than once in that list, which may indicate a recursive call to some function. In addition, we always remember the last call site. In practice, the list of call sites is an additional parameter to the abstract interpreter. Following [50] again, we use this list of call sites to decide when widening on the environments should be performed: it is performed only when eval is called on a *maximal* list of call sites. The same list of call sites is also used to derive dynamic exception names and abstract pointers (see §6.4 and §6.5).

Flow sensitivity. Our abstract interpreter is able to exploit information that is learned when a branch in a match is taken, or when branching on an arithmetic test. For example, in the program match (x, y) with $(\mathsf{None}, \_) \Rightarrow x \mid \_ \Rightarrow t$, our analyser is able to refine the possible environments, by taking into account that $x = \mathsf{None}$ in the first branch, and that this first branch necessarily returns the value None. This is done by performing a *backward analysis* of the scrutinee (x, y). This backward analysis infers an over-approximation of the environment, knowing that the scrutinee successfully matched against the pattern $(\mathsf{None}, \_)$.

Dynamic partitioning. Finally, we have employed a form of *dynamic partitioning* to avoid conflating some analyses results, that could degrade precision. Based on a notion of *similarity* on the shapes of abstract values found in environments, we decide whether to conflate contexts or not. The technique is inspired by the *silhouettes* used in shape analysis [39].

6.2 Transformation of Typed OCaml ASTs

The actual language that our interpreter takes as input is more complex than the one we presented in §3, but undoubtedly simpler than the OCaml AST. The main differences between our intermediate language and the OCaml AST, is that we deal with only one construct for pattern matching, and only one construct for exception handling, and that those two constructs implement orthogonal features in our language. This is in contrast with OCaml's `try t1 with p -> t2` and `match t with p1 -> t1 | exception p2 -> t2`, that conflate pattern matching with exception handling. The transformation into our two constructs is mostly straightforward, and greatly simplifies the job of the static analyser.

Our intermediate language makes the evaluation order explicit using let bindings. While the evaluation order in OCaml is generally unspecified, we did our best to mimic the choices that the OCaml compiler makes.

We added specific application nodes for OCaml primitives. To ensure they are called with the correct arity, we inserted λ-abstractions when they were partially applied, or additional application nodes when they were given more arguments than expected. We also handled specifically the short-circuiting primitives on boolean expressions && and ||, as they change the evaluation order.

We kept the n-ary application nodes of the OCaml AST (instead of the binary applications from §3), as this is important for the semantics of labelled/optional function arguments. Nevertheless, the transformation from the OCaml AST into our intermediate language needed a lot of care and effort. In particular, missing labelled arguments required the insertion of λ-abstractions, which can be particularly subtle when interacting with optional arguments.

6.3 Pattern Disambiguation

The last major difference between OCaml and our intermediate language is the exhaustive and non-ambiguous requirements on pattern matching. These properties not only simplify the semantics of our intermediate language, but also facilitate the analysis of programs. Indeed, each branch of the pattern-matching can be analysed *independently* of the other ones, whereas in OCaml, branches must be considered in order, until one pattern matches the inspected value. The OCaml type-checker still provides warnings to verify the *utility* of each branch and the *exhaustiveness* of the overall pattern matching.

Enforcing exhaustive and non-ambiguous pattern matchings in OCaml would require to use of cumbersome patterns, and, furthermore, it is not always possible to write such patterns in OCaml. It is, indeed, allowed to match on values whose types may have an infinity of constructors, *e.g.*, arrays, strings, or extensible variant types (see §6.4 for details). To reach these requirements, we extend the language of patterns with a complement $p \setminus q$ [8]. A value v matches a pattern $p \setminus q$ if and only if it matches p but not q. In an ordered pattern matching $\mathsf{match}\ t\ \mathsf{with}\ p_1 \Rightarrow u_1 \mid \cdots \mid p_n \Rightarrow u_n$, we can express that the value v of the term t matches the i^{th} pattern, unambiguously. It suffices to add that v does not match any of the preceding patterns p_j with $j < i$, *i.e.*, v matches $p_i \setminus (\Sigma p_j) \prec\!\!\prec v$.

The method presented in [8] shows how to solve the disambiguation problem [32]. It relies on the notion of pattern semantics $[\![p]\!]$ that is the set of values matched by a pattern: $[\![p]\!] = \{v \in \mathbb{V} \mid p \prec\!\!\prec v\}$. The idea is to reduce any pattern p into a purely disjunctive pattern q, *i.e.*, a pattern containing no complements $\setminus$, while preserving its semantics : $[\![p]\!] = [\![q]\!]$. The reduction relies on rewriting rules that correspond to algebraic laws of set theory: a constructor c behaves like a labelled cartesian product, the disjunction $+$ like set union, and the complement $\setminus$ like set difference. Note that the pattern language proposed in §3 conflates the different forms of OCaml constructors (constructor variant, polymorphic variant, records, arrays and tuples) as they behave similarly *w.r.t.* to their semantics.

In order to *fully* reduce a pattern, the method also relies on the observation that a variable x_τ of a variant type τ must be matched by a value whose head is a constructor of the type τ. Therefore, the semantics of this variable x_τ can be described as the union of semantics of all constructor instances of τ: $[\![x_\tau]\!] = \bigcup_{c \in \mathcal{C}_\tau} [\![c(z_1, \ldots, z_n)]\!]$, where $\mathcal{C}_\tau$ is the finite set of constructors of co-domain τ. Similarly, the utility [40] approach, implemented in the OCaml compiler, relies on the ability to enumerate all the constructors of a type to provide a non-ambiguous description of the useful patterns. For types that may not be finitely described, the semantic approach can still be used to partially reduce the *complements* [7]. We keep *anti-patterns*—patterns of the form $x \setminus q$ where q contains no complements—when there exists a value v such that $x \setminus q \prec\!\!\prec v$.

Finally, to guarantee the exhaustiveness of pattern matching, it suffices to add a rule $z \setminus (p_1 + \cdots + p_n) \Rightarrow \mathsf{raise\ Match\_failure}$ when necessary. Again, generating such a non-ambiguous rule, for data types that may not be finitely described, is only possible thanks to pattern complements.

6.4 Dynamic Exceptions

The exception type in OCaml is an *extensible variant type*: it can be *dynamically* extended with new variant constructors. This means that new exception con-

$$t ::= \ldots \mid \mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t \mid \mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t$$
$$v ::= \ldots \mid d \mid d(v_1, \ldots, v_k)$$

$$\frac{E(\overline{c}) = d}{S; E \vdash \overline{c}(x_1, \ldots, x_k) \Downarrow_{\mathsf{val}} S; d(E(x_1), \ldots, E(x_n))}\ \textsc{DynamicConstruct}$$

$$\frac{S \uplus \{d\}; E, \overline{c} \mapsto d \vdash t \Downarrow_m S'; v}{S; E \vdash \mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t \Downarrow_m S'; v}\ \textsc{LetException}$$

$$\frac{S; E, \overline{b} \mapsto d \vdash t \Downarrow_m S'; v \qquad E(\overline{c}) = d}{S; E \vdash \mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t \Downarrow_m S'; v}\ \textsc{RebindException}$$

$$\begin{array}{ll} \mathsf{A} ::= \{\ldots;\ \mathsf{names} : \mathsf{V}\} \mid \alpha \mid \mu\alpha.\mathsf{A} & \text{(Abstract value)} \\ \mathsf{V} ::= \{\overline{(c, \delta)}\} & \text{(Abstract names)} \end{array}$$

$$\begin{aligned} [\![\mathsf{let\ exception}\ \overline{c}\ \mathsf{of}\ \tau_1 * \cdots * \tau_n\ \mathsf{in}\ t]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}, \overline{c} \mapsto \{\mathsf{names} = (c, \delta)\}} \\ [\![\mathsf{let\ exception}\ \overline{b} = \overline{c}\ \mathsf{in}\ t]\!]^{\mathsf{eval}}_{\mathsf{E}} &= [\![t]\!]^{\mathsf{eval}}_{\mathsf{E}, \overline{b} \mapsto \mathsf{E}(\overline{c})} \end{aligned}$$

Fig. 3. Changes to support dynamic exception naming (excerpts).

structors are dynamically generated during the execution of programs. Although this section focuses on the exception type, the techniques we present apply to any extensible variant type as well.

To model the dynamic behaviour of type extension, we introduce dynamic constructors, written $\overline{c}$, that, unlike static constructors c, are dynamically associated to a variant name d during the evaluation. We update the language of §3 and its semantics to support these dynamic constructors (Figure 3).

The let exception $\overline{c}$ of $\tau_1 * \cdots * \tau_n$ in t construct defines the new exception constructor $\overline{c}$, that is dynamically bound to a fresh variant name in the sub-term t. The *exception alias* construct let exception $\overline{b} = \overline{c}$ in t defines the exception constructor $\overline{b}$, that is bound in the sub-term t to the variant name of $\overline{c}$. Constructed values can now have a dynamic variant name as their head constructor.

To account for the generative aspect of dynamic constructors, the evaluation rules now carry an execution state S, that contains the set of the already generated variant names. These are akin to the *time-stamps* from the CFA literature [25,44], that are used to allocate data in memory locations. In the analysis, we use an over-approximation δ of the list of call sites—that we used already in §6.1 to control the widening strategy—to give abstract names (c, δ) to dynamic constructors.

Finally, as the variant name of an exception constructor is resolved dynamically, the pattern matching relation depends on the evaluation environment E: $\overline{c}(p_1, \ldots, p_n) \prec\!\!\prec d(v_1, \ldots, v_n)$ if and only if $E(\overline{c}) = d$, and $p_i \prec\!\!\prec v_i$ for all $i \in [1, n]$.

As the exception type is extensible, a *finite* number of constructor patterns never forms an exhaustive set of patterns for the exception type. Therefore, the utility approach on pattern matching [40] used in OCaml for exhaustiveness checking cannot provide an exhaustive list of *non-ambiguous* counter-examples: that list is not known statically. In contrast, the disambiguation approach from §6.3 is particularly well suited to such types, by leveraging *anti-patterns* [7]. Moreover,

$$
\begin{array}{ll}
t ::= \dots \mid \{f_1 = x_1; \dots ; f_n = x_n\} \mid x.f \mid x.f \leftarrow y & \\
v ::= \dots \mid \ell & \text{(Heap locations)} \\
S ::= \{\ell_1 \mapsto r_1;\ \dots ;\ \ell_n \mapsto r_n\} & \text{(Memory heaps)} \\
r ::= \{f_1 \mapsto v_1;\ \dots ;\ f_n \mapsto v_n\} & \text{(Record blocks)}
\end{array}
$$

$$
\frac{\ell \notin \operatorname{dom} S \qquad S' = S, \ell \mapsto \{f_1 \mapsto E(x_1); \dots ; f_n \mapsto E(x_n)\}}{S; E \vdash \{f_1 = x_1; \dots ; f_n = x_n\} \Downarrow_{\mathsf{val}} S'; \ell}\ \textsc{Alloc}
$$

$$
\frac{E(x) = \ell \qquad S(\ell) = \{f_1 \mapsto v_1; \dots ; f_n \mapsto v_n\} \qquad 1 \leq i \leq n}{S; E \vdash x.f_i \Downarrow_{\mathsf{val}} S; v_i}\ \textsc{GetField}
$$

$$
\frac{\begin{array}{c} E(x) = \ell \qquad S(\ell) = \{f_1 \mapsto v_1; \dots ; f_n \mapsto v_n\} \\ 1 \leq i \leq n \qquad S' = S, \ell \mapsto \{f_1 \mapsto v_1; \dots ; f_i \mapsto E(y); \dots ; f_n \mapsto v_n\} \end{array}}{S; E \vdash x.f_i \leftarrow y \Downarrow_{\mathsf{val}} S'; ()}\ \textsc{SetField}
$$

$$
\begin{array}{ll}
\mathsf{A} ::= \{\ \dots ;\ \mathsf{locs} : \{\ell^\sharp_1, \dots , \ell^\sharp_n\}\} & \text{(Abstract locations in abstract values)} \\
h^\sharp ::= \{\ell^\sharp_1 \mapsto r^\sharp_1;\ \dots ;\ \ell^\sharp_n \mapsto r^\sharp_n\} & \text{(Abstract heaps)} \\
r^\sharp ::= \{f_1 \mapsto \mathsf{A}_1;\ \dots ;\ f_n \mapsto \mathsf{A}_n\} & \text{(Abstract record blocks)}
\end{array}
$$

Fig. 4. Changes to support mutable records (excerpts).

the equality of two exception constructors $\overline{b}$ and $\overline{c}$ of the same arity can only be resolved dynamically. Therefore, there is no way to statically prove, or disprove, the utility of a pattern $\overline{b}(q_1, \dots , q_n)$ against a pattern $\overline{c}(p_1, \dots , p_n)$. On the other hand, in our pattern formalism, we can simply write $\overline{b}(q_1, \dots , q_n) \setminus \overline{c}(p_1, \dots , p_n)$ to guarantee the non-ambiguity between the two.

6.5 Mutable Records and Global State

OCaml supports *mutable* records. While immutable records can be modelled in the programming language of §3 in the form of *constructs*—an immutable record is a variant with a single case—mutable records require extending the semantics with a global memory heap S (Figure 4).

Heaps are maps from memory locations ℓ to record blocks. Record blocks are structured memory blocks, that contain values for all the registered fields of the record. The standard notion of *reference* can be modelled as a mutable record with a single field. This is exactly how the type of references is defined in OCaml.

We adapt the big-step semantics in a standard way, so that it takes a heap as input and returns an updated heap as output. The evaluation rules for record creation, access, and update, either query or modify the memory heap as expected.

OCaml features pattern matching on mutable records. We adapt the rules for pattern matching, so that matching on a mutable record first queries the memory heap to retrieve the values for the fields of the record, before matching continues.

To analyse programs that involve mutable records, we add a new field to abstract values, that contains the possible *abstract locations* $\ell^\sharp$ a value might be equal to. Abstract locations denote sets of concrete locations. Similarly to the dynamic extension constructors of §6.4, fresh abstract locations are chosen by following a naming scheme that is based on the abstract call stack.

The abstract interpreter is easily adapted to support global state, by lifting the abstract exception monad to the state monad, where states are abstract heaps. Abstract heaps map abstract locations to abstract record blocks, that themselves map record fields to abstract values. The operations on abstract heaps and the transfer functions on records are standard, and elided from the presentation.

6.6 Modules and Functors

The OCaml language includes an expressive module system [36], that supports hierarchical structures, higher-order functors, and first-class modules. In this section, we give the reader the main insights for the analysis of OCaml modules.

First, we consider an *untyped* semantics of modules, *i.e.*, we do not propagate type information. In particular, we do not take type abstraction boundaries into account. We carefully to keep track of module *coercions*, however: signature ascriptions may have, indeed, a computational content, as they can remove some module fields. Coercions are automatically applied at functor applications to "reshape" the functor argument. Coercions distribute on functors, contravariantly on their formal arguments, and covariantly on their results.

Embracing further the untyped nature of our approach, we made the choice of having a *single class of values*, that comprises both values from the core language and values for module structures and functor closures. This simplifies both the concrete semantics (for example, transfers from the module language to the core language and back are *no-ops*), and the design of the abstract domain. As we sketched in the previous sections, it suffices to add new fields to abstract values to describe the possible structures and functor closures.

We represent structures as unordered records, *i.e.*, maps from field names to values. Functor closures hold the functor code, an environment, and coercions for the argument and the result, that shall be applied when the functor is called.

Importantly, the support of dynamic exceptions (§6.4) was *required* to support functors, since an exception might be declared in a functor's body: this leads to the creation of a fresh exception every time this functor is instantiated.

The analysis functions for the core language and the module language, of types $\mathbb{T} \to \mathbb{E} \to \mathbb{A}$ and $\mathbb{M} \to \mathbb{E} \to \mathbb{A}$, are mutually recursive. Still, the approach of using a fixpoint solver to define our abstract interpreter remains applicable. The two functions can be transformed into a single function of type $(\mathbb{T}+\mathbb{M}) \to \mathbb{E} \to \mathbb{A}$, then given to the solver, and split back into two functions. Our untyped approach was again crucial, as we could keep a single type of abstract values, and a single type of abstract environments, which made the previous transformation possible.

7 Experiments

We tested our prototype analyser for OCaml programs on 290 programs, that range from small, manually written programs, to larger examples extracted

Table 1. Experiments: size of the programs, analysis time (with minimisation disabled, and enabled). They are sorted by program decreasing size.

Program	Size (LoC)	Analysis (w/o minim.)	Analysis (w/ minim.)
`heintze_mcallester_1000`	4002	0.2 s	0.2 s
`boyer`	1292	26 m	57 m
`kb`	552	1.2 s	1.4 s
`map_merge`	152	4.5 s	5.6 s
`sliding_window`	122	4.4 s	5.8 s
`skolemize`	82	38 m	2.9 s
`negative_normal_form`	64	40 m	4.6 s
`red_black_trees2`	64	0.5 s	1.0 s
`church`	20	0.05 s	0.07 s
`sieve`	19	0.01 s	0.01 s
`tak_cps`	8	0.04 s	0.04 s
`tak`	4	< 0.01 s	0.01 s
`mc91_cps`	4	< 0.01 s	< 0.01 s
`mc91`	2	< 0.01 s	< 0.01 s

from the literature or from the OCaml compiler's test suite. The test programs include some classic functions such as the factorial program from §2, Takeuchi's function, McCarthy's 91 function, fixpoint combinators, programs that compute over church numerals, transformations of abstract syntax trees for arithmetic expressions or logical formulas, and the algorithm for Knuth-Bendix completion of rewriting systems. The test suite covers a large array of coding styles, *e.g.*, direct style, continuation-passing style, monadic style, or imperative style, and exhibits different language features, *e.g.*, assertions, exception-based control flow, GADTs and non-regular types, polymorphic recursion, second-order polymorphism, *etc.*

We present in Table 1, a selection of the test results on some key examples. The complete test results are reproducible via the companion artefact [35]. The experimental results are encouraging, both in terms of performance and precision.

In terms of precision, our analyser infers the best achievable abstract values on several programs: For McCarthy's 91 function `mc91`, the result is shown to be greater than 91; for the skolemisation of logical formulas `skolemize`, the analyser correctly infers the form of returned terms, *i.e.*, they cannot contain existential quantifiers. For other programs, the analyser only infers an over-approximation: for the `red_black_tree` program, it correctly infers the general shape of trees, but cannot infer the structural invariant that no red node has red children.

The `map_merge` example calls the `Map.Make` functor of finite maps from the standard library, builds several maps, and calls the `merge` function on those maps, that merges the maps. The `merge` function has the following signature:

```
val merge: (key -> 'a option -> 'b option -> 'c option) ->
           'a M.t -> 'b M.t -> 'c M.t
```

Its first argument specifies what should be done when a key/value pair is found in one of the maps, or in both. This argument is *never* called for keys that are absent in both maps, *i.e.*, the case where the second and third arguments are both equal to `None` is *unreachable*. OCaml programmers often write `assert false` in the corresponding pattern matching branch. The analyser infers that the `Assertion_failure` exception is never raised, which means that this branch cannot be reached. The analyser cannot show, however, that every assertion present in `Map.Make` is satisfied: in the re-balancing function for pseudo-balanced trees, assertion failures are reported, because the analyser cannot infer that the heights that are recorded in the trees are strictly positive.

In terms of performance, most examples, and even some large programs, are analysed in a couple of seconds, or in less than a second. In contrast, some examples like `boyer` need approximately one hour for the analysis to terminate. `boyer` is a tautology checker, that is run on a large formula (its definition takes about 1000 lines). This formula, of mutable type, requires the creation of several hundreds of abstract pointers, which makes abstract operations on abstract heaps very costly. If we reduce context sensitivity to "the *last* call site", fewer abstract pointers are created, and the analysis completes in 31 s. This suggests that context sensitivity choices for naming abstract pointers need further investigation.

Our experiments show that the minimisation of abstract values during widening and unions (§4.2) may impact performance positively or negatively. For instance, for AST transformations like `skolemize` and `negative_normal_form`, minimisation decreases the analysis time from about 45 m down to a few seconds. For `boyer`, however, minimisation incurs a heavy cost, as it doubles the analysis time. Further investigations are needed to reduce the cost of minimisation.

8 Related Work

The static detection of uncaught exceptions for ML programs has been the topic of many related work. We only discuss a selection of them, and some results on static analysis of functional programs that are also relevant to the current work.

Set Constraints. Several static analyses for functional programs were based *set constraints* [21]. The principle is to transform a program into a constraint, that features unions, intersections, negations, and a form of conditional constraint. Then, the constraint is simplified and given to a solver, from which the analysis result is obtained. Fähndrich and his coauthors built a exception analysis tool that infers types and effects for SML programs [14,15] using the BANE constraint analysis engine, using a mix of set constraints and type constraints.

Type and Effect Systems. Pessaux and Leroy have developed `ocamlexc` [38,54,55], a tool that detects uncaught exceptions in OCaml programs. They use a *type and effect* system to analyse programs modularly. Their analyser extends unification-based type inference, and makes use of *row variables* [57] and polymorphism to produce precise types for functions. They type variants *structurally* using equi-recursive types. Recursion may also occur through the effect annotations on arrow types. They also describe an algorithm to improve the accuracy of their analysis, that uses *polymorphic recursion* for row variables. The programming language Koka [33] also leverages row variables to type algebraic effects. Recently, de Vilhena and Pottier [62] devised a type system based on row variables for a language that supports the dynamic creation of algebraic effects.

Control-Flow Analyses. An important family of analyses for higher-order programs are control-flow analyses (CFA) [60,65,51,45,19]. The goal of CFA is to determine which functions might be called at a call site, and on which arguments. CFA can be expressed as instances of abstract interpretation [46,44,47,50]. CFA can easily be extended to analyse exceptions. Yi developed an abstract interpreter that detects uncaught exceptions in SML [68,67,66]. It implements an analysis that is close to a 0-CFA analysis extended to support exceptions.

Abstract Domains in CFA. Most previous work on CFA share a common representation for abstract values: Although they need to represent some inductively defined sets, they refrain from using a native device to express fixpoints, such as our μ constructor. Instead, cyclic definitions are encoded using *indirections through abstract pointers*, that point to an *abstract heap*. For example, the inductive set of continuations from §2 is expressed as follows in CFA domains:

$$\begin{array}{ll} \{\mathsf{funs} : \{(\lambda x.\, x) \mapsto \{\};\ (\lambda x.\, k\ (x * i)) \mapsto \{i \mapsto p_i; k \mapsto p_k\}\}\} & \\ \text{where:}\quad \hat{h}(p_i) = \{\mathsf{ints} : [1, +\infty]\} & \\ \qquad\qquad\ \hat{h}(p_k) = \{(\lambda x.\, x) \mapsto \{\};\ (\lambda x.\, k\ (x * i)) \mapsto \{i \mapsto p_i; k \mapsto p_k\}\} & \end{array}$$

In this abstract value, the closures' environments contain the pointers p_i and p_k, that are defined in the abstract heap $\hat{h}$. This abstract heap contains a cycle, since p_k is used in the definition of the abstract value pointed by p_k. This is in contrast to our approach, where we make use of μ nodes to introduce cycles directly, without referring to a heap. We only use the abstract heap for mutable data. In CFA domains, all data (constructs, closures, *etc.*) are "abstractly allocated" in the abstract heap, regardless of whether they are mutable or not.

A benefit of the approach with heap indirections is that abstract values have a bounded height, and cycles need no special treatment: The equality of abstract pointers is used to compute on abstract values. While this makes the operations of CFA abstract domains easy to define, using pointer names limits drastically the detection of semantically equivalent values. We argue that our approach allows to detect more semantics inclusions, therefore decreasing the number of iterations of the analysers, at the cost of more complex abstract domain operations.

Tree Grammars. Several analyses for functional languages have been defined using *tree grammars*. For example, Reynolds [58] defined an analysis for pure first-order LISP using *data sets*, *i.e.*, tree grammars that denote the possible outputs of function symbols. Extended tree grammars, *i.e.*, grammars with *selectors* of the form $X \to Y.hd$, have been used by Jones and his coauthors to analyse full LISP

[28], and, later, strict and lazy λ-calculi [26,27]. From a λ-term, they produce tree grammars with selectors, that denote the possible inputs and outputs of function symbols. Selectors can then be eliminated in order to simplify the grammars. *Deterministic* tree grammars have been identified as an abstract domain to recast analyses based on set constraints into the abstract interpretation framework [10].

Tree Automata. Generalising string automata, tree automata are an established formalism to represent sets of trees. They have been used to define static analysers for term-rewriting systems (TRSs) [3] and higher-order programs [20]. They have been extended to *lattice tree automata* to support arbitrary non-relational abstract domains at their leaves [17,18], and improve the performance of analysers for TRSs. Recently, tree automata were combined with relational numeric abstract domains [29], to express *relations* between scalar data contained in trees. Recent work report on the design of relational domains for algebraic data types [2,61].

Cyclic Abstract Domains. Type graphs [22] are a form of deterministic tree grammars, that are represented as cyclic graphs with *no sharing*, *i.e.*, trees with cycles. They have been used to analyse Prolog programs. We used a similar graph-based representation as an intermediate form to compute union, intersection and widening. We use, however, a term-based representation with binders as our main representation, as it allows easy and efficient hash-consing and memoisation [13]. Our widening operator (§4.2) is inspired by the one from type graphs.

Mauborgne [42,43,41] studied graph-based abstract domains for sets of trees, and defined ways to have minimal, canonical representations of such abstract values. Using Mauborgne's structures natively could improve our analyser's performance, as we could avoid translating back and forth from terms to graphs.

Finally, *recursive types* [56] were a strong inspiration for the abstract domain of §4. Recursive types have been thoroughly studied in the context of subtyping [16,31,1], where polynomial algorithms have been devised to decide inclusion. They proceed by translating types into variants of tree automata, that can also deal with the contravariance of arrow types.

Fixpoint Solvers. To the best of our knowledge, Le Charlier and Hentenryck [6] were the first to exploit a dynamic fixpoint solvers to define static analysers. They used the *top-down solver* to analyse Prolog programs. The same approach has been followed for the Goblint static analyser for C programs [64,59], and for the analysis of WebAssembly programs [4]. Recent work introduced combinators to define dynamic fixpoint solvers in a modular manner [30]. Several dynamic fixpoint solvers have been successfully formally verified [24,63].

9 Conclusive Remarks and Future Work

We have introduced a λ-calculus that features pattern matching primitives and exception handling, in which exceptions are first-class citizens. We have presented a static analysis for this language, in the form of a monadic abstract interpreter, that can be used as an effective static analyser. This analyser detects uncaught exceptions, and provides a description of the values that a program may return. The abstract interpreter relies on a generic abstract domain, that is parameterised over a domain for scalars, and that can represent *regular sets* of values of our programming language. This is achieved by a fixpoint constructor in the syntax of abstract values, that denotes an inductive set of values.

The abstract interpreter is defined in an *open recursive style*, where the recursive knot is tied by calling a *dynamic fixpoint solver*. Importantly, the analyser does not call the solver for every recursive call: it performs standard recursive calls on strict sub-terms, but calls the solver to analyse function calls.

Based on this approach, we implemented a static analyser for OCaml programs. We presented some extensions of our formalism to support several core features of OCaml, including dynamic generation of exceptions, mutable records, the module system. Our analyser starts with transforming the OCaml typed AST into a simpler language where evaluation order is explicit. This transformation required a lot of care and demanded a substantial implementation effort. One key aspect of this transformation is the disambiguation of pattern matching, as we chose to work with an *exhaustive and non-ambiguous* pattern matching primitive in order to simplify the analysis of programs.

Our experiments on 290 OCaml programs show some encouraging results, both in terms of performance and precision. Still, some improvements are needed for the analysis to be applicable to larger code bases. In particular, the minimisation of abstract values requires some more study and fine tuning: while it plays a crucial role to analyse some examples in a reasonable time, it can also severely undermine the analyser's performance in some other cases.

At the moment, the analyser can deal with whole programs only. To analyse libraries more modularly, we plan to experiment with generating abstract values that over-approximate the inputs of a library's function, based on their types. In the near future, we also plan to extend the analyser with OCaml features that are yet to be supported (*e.g.*, arrays, laziness, floats, objects, recursive modules, interactions with the operating system, *etc.*), most of which will require substantial formalisation and implementation efforts. Recently introduced features, such as algebraic effects and one-shot continuations, are also on our agenda, and are likely to raise interesting challenges.

Finally, we hope that our abstract interpreter can be extended to perform other kinds of static analyses for OCaml programs, such as a purity analysis, or the detection of whether the behaviour of a program might depend on the *order of evaluation*. We would also like our implementation to serve as a basis for experimenting with recent relational domains for trees and scalars [29,61,2], and with relational analyses of functional programs [49].

Data-Availability Statement. The companion artefact [35] is hosted on the Zenodo platform and referenced by the DOI `10.5281/zenodo.10457925`.

References

1. Amadio, R.M., Cardelli, L.: Subtyping recursive types p. 575–631 (9 1993). https://doi.org/10.1145/155183.155231
2. Bautista, S., Jensen, T., Montagu, B.: Lifting Numeric Relational Domains to Algebraic Data Types. In: Singh, G., Urban, C. (eds.) Static Analysis. pp. 104–134. Springer Nature Switzerland, Cham (2022)
3. Boichut, Y., Genet, T., Jensen, T., Roux, L.L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. In: Lecture Notes in Computer Science, pp. 48–62. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_6
4. Brandl, K., Erdweg, S., Keidel, S., Hansen, N.: Modular Abstract Definitional Interpreters for WebAssembly. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.5
5. Charguéraud, A.: The Locally Nameless Representation. Journal of Automated Reasoning **49**(3), 363–408 (May 2011). https://doi.org/10.1007/s10817-011-9225-2
6. Charlier, B.L., Van Hentenryck, P.: A Universal Top-Down Fixpoint Algorithm. Tech. rep., USA (1992), ftp://ftp.cs.brown.edu/pub/techreports/92/cs92-25.pdf
7. Cirstea, H., Lermusiaux, P., Moreau, P.E.: Static analysis of pattern-free properties. In: Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming, PPDP 2021. pp. 9:1–9:13. ACM (sep 2021). https://doi.org/10.1145/3479394.3479404
8. Cirstea, H., Moreau, P.: Generic Encodings of Constructor Rewriting Systems. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, PPDP 2019. pp. 8:1–8:12. ACM (oct 2019). https://doi.org/10.1145/3354166.3354173
9. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
10. Cousot, P., Cousot, R.: Formal Language, Grammar and Set-constraint-based Program Analysis by Abstract Interpretation. In: Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95. ACM Press (1995). https://doi.org/10.1145/224164.224199
11. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). Proc. ACM Program. Lang. **1**(ICFP), 12:1–12:25 (2017). https://doi.org/10.1145/3110256
12. Fecht, C., Seidl, H.: A Faster Solver for General Systems of Equations. Sci. Comput. Program. **35**(2), 137–161 (1999). https://doi.org/10.1016/S0167-6423(99)00009-X
13. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 workshop on ML. ACM (sep 2006). https://doi.org/10.1145/1159876.1159880
14. Fähndrich, M., Aiken, A.: Tracking down exceptions in Standard ML programs. techreport 98-996, University of California at Berkeley, Computer Science Division (1998)
15. Fähndrich, M., Forster, J.S., Aiken, A., Cu, J.: Tracking down Exceptions in Standard ML Programs. techreport UCB/CSD-98-996, University of California, Computer Science Division (EECS), Berkeley, California 94720 (Feb 1998), https://theory.stanford.edu/~aiken/publications/papers/tr98.pdf
16. Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive Subtyping Revealed. Journal of Functional Programming **12**(6), 511–548 (2002). https://doi.org/10.1017/S0956796802004318

17. Genet, T., Gall, T.L., Legay, A., Murat, V.: A Completion Algorithm for Lattice Tree Automata. In: Konstantinidis, S. (ed.) Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7982, pp. 134–145. Springer (2013). https://doi.org/10.1007/978-3-642-39274-0_13
18. Genet, T., Le Gall, T., Legay, A., Murat, V.: Tree Regular Model Checking for Lattice-Based Automata. In: CIAA - 18th International Conference on Implementation and Application of Automata. LNCS, vol. 7982. Springer, Halifax, Canada (Jul 2013)
19. Gilray, T., Lyde, S., Adams, M.D., Might, M., Horn, D.V.: Pushdown control-flow analysis for free. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 691–704. ACM (2016). https://doi.org/10.1145/2837614.2837631
20. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. Proc. ACM Program. Lang. **4**(ICFP), 112:1–112:29 (2020). https://doi.org/10.1145/3408994
21. Heintze, N., Jaffar, J.: Set constraints and set-based analysis. In: Lecture Notes in Computer Science, pp. 281–298. Springer Berlin Heidelberg (1994). https://doi.org/10.1007/3-540-58601-6_107
22. Hentenryck, P.V., Cortesi, A., Charlier, B.L.: Type analysis of Prolog using type graphs. The Journal of Logic Programming **22**(3), 179–209 (Mar 1995). https://doi.org/10.1016/0743-1066(94)00021-w
23. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. Journal of Functional Programming **16**(02), 197 (nov 2005). https://doi.org/10.1017/s0956796805005769
24. Hofmann, M., Karbyshev, A., Seidl, H.: Verifying a Local Generic Solver in Coq. In: Cousot, R., Martel, M. (eds.) Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6337, pp. 340–355. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_21
25. Horn, D.V., Might, M.: Abstracting Abstract Machines. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10. ACM Press (2010). https://doi.org/10.1145/1863543.1863553
26. Jones, N.D.: Flow Analysis of Lambda Expressions. DAIMI Report Series **10**(128) (Jan 1981). https://doi.org/10.7146/dpb.v10i128.7404
27. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science **375**(1-3), 120–136 (May 2007). https://doi.org/10.1016/j.tcs.2006.12.030
28. Jones, N.D., Muchnick, S.S.: Flow Analysis and Optimization of LISP-like Structures. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '79. ACM Press (1979). https://doi.org/10.1145/567752.567776
29. Journault, M., Miné, A., Ouadjaout, A.: An Abstract Domain for Trees with Numeric Relations. 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. 724–751. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_26

30. Keidel, S., Erdweg, S., Hombücher, T.: Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. Proceedings of the ACM on Programming Languages **7**(ICFP), 955–981 (aug 2023). https://doi.org/10.1145/3607863
31. Kozen, D., Palsberg, J., Schwartzbach, M.I.: Efficient Recursive Subtyping. Mathematical Structures in Computer Science **5**(1), 113–125 (Mar 1995). https://doi.org/10.1017/s0960129500000657
32. Krauss, A.: Pattern minimization problems over recursive data types. In: Hook, J., Thiemann, P. (eds.) Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP '08. pp. 267–274. ACM (sep 2008). https://doi.org/10.1145/1411204.1411242
33. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. ACM (jan 2017). https://doi.org/10.1145/3009837.3009872
34. Lermusiaux, P., Montagu, B.: Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation (Extended Version). Research report, Inria (Jan 2024), https://inria.hal.science/hal-04410771
35. Lermusiaux, P., Montagu, B.: Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation: Software Artefact (Jan 2024). https://doi.org/10.5281/zenodo.10457925
36. Leroy, X.: A Modular Module System. Journal of Functional Programming **10**(3), 269–303 (2000). https://doi.org/10.1017/S0956796800003683
37. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml System, Documentation and User's Manual – Release 5.1. INRIA (Nov 2023), https://v2.ocaml.org/releases/5.1/htmlman/index.html
38. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. ACM Transactions on Programming Languages and Systems **22**(2), 340–377 (2000). https://doi.org/10.1145/349214.349230
39. Li, H., Berenger, F., Chang, B.E., Rival, X.: Semantic-directed clumping of disjunctive abstract states. 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. 32–45. ACM (2017). https://doi.org/10.1145/3009837.3009881
40. Maranget, L.: Warnings for pattern matching. Journal of Functional Programming **17**(3), 387–421 (2007). https://doi.org/10.1017/S0956796807006223
41. Mauborgne, L.: Representation of Sets of Trees for Abstract Interpretation. phdthesis, École Polytechnique (Nov 1999), https://www.di.ens.fr/~mauborgn/publi/t.pdf
42. Mauborgne, L.: An Incremental Unique Representation for Regular Trees. Nordic Journal of Computing **7**(4), 290–311 (Dec 2000), https://software.imdea.org/~mauborgn/publi/njc7.pdf
43. Mauborgne, L.: Improving the representation of infinite trees to deal with sets of trees. In: Smolka, G. (ed.) Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1782, pp. 275–289. Springer (2000). https://doi.org/10.1007/3-540-46425-5_18
44. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys **44**(3), 10:1–10:33 (2012). https://doi.org/10.1145/2187671.2187672
45. Midtgaard, J., Jensen, T.: A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In: Static Analysis, pp. 347–362. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_23

46. Midtgaard, J., Jensen, T.P.: Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09. ACM Press (2009). https://doi.org/10.1145/1596550.1596592
47. Midtgaard, J., Jensen, T.P.: Control-Flow Analysis of Function Calls and Returns by Abstract Interpretation. Information and Computation **211**, 49–76 (Feb 2012). https://doi.org/10.1016/j.ic.2011.11.005
48. Milner, R.: A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences **17**(3), 348–375 (dec 1978). https://doi.org/10.1016/0022-0000(78)90014-4
49. Montagu, B., Jensen, T.P.: Stable Relations and Abstract Interpretation of Higher-order Programs. Proc. ACM Program. Lang. **4**(ICFP), 119:1–119:30 (2020). https://doi.org/10.1145/3409001
50. Montagu, B., Jensen, T.P.: Trace-Based Control-Flow Analysis. 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, 20211. pp. 482–496. ACM (2021). https://doi.org/10.1145/3453483.3454057
51. Nielson, F., Nielson, H.R.: Interprocedural Control Flow Analysis. In: Programming Languages and Systems, pp. 20–39. Springer Berlin Heidelberg (1999). https://doi.org/10.1007/3-540-49099-x_3
52. Okasaki, C.: Red-Black Trees in a Functional Setting. J. Funct. Program. **9**(4), 471–477 (1999), http://journals.cambridge.org/action/displayAbstract?aid=44273
53. Okasaki, C., Gill, A.: Fast Mergeable Integer Maps. In: Morriset, G. (ed.) Proceedings of the 1998 ACM SIGPLAN workshop on ML. pp. 77–86 (Sep 1998)
54. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM (jan 1999). https://doi.org/10.1145/292540.292565
55. Pessaux, F.: Détection statique d'exceptions non rattrapées en Objective Caml. Ph.D. thesis, Université Paris 6 (1999), http://www.theses.fr/1999PA066398
56. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, Mass (2002)
57. Rémy, D.: Type checking records and variants in a natural extension of ML. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89. ACM Press (1989). https://doi.org/10.1145/75277.75284
58. Reynolds, J.C.: Automatic Computation of Data Set Definitions. Information Processing **68**, 456–461 (1969)
59. Seidl, H., Vogler, R.: Three Improvements to the Top-Down Solver. In: Sabel, D., Thiemann, P. (eds.) Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018. pp. 21:1–21:14. ACM (2018). https://doi.org/10.1145/3236950.3236967
60. Shivers, O.: The Semantics of Scheme Control-Flow Analysis. In: Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 190–198. PEPM '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/115865.115884
61. Valnet, M., Monat, R., Miné, A.: Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récursifs. In: JFLA 2023-34èmes Journées Francophones des Langages Applicatifs. pp. 210–241 (2023)

62. de Vilhena, P.E., Pottier, F.: A Type System for Effect Handlers and Dynamic Labels. In: Programming Languages and Systems, pp. 225–252. Springer Nature Switzerland (2023). https://doi.org/10.1007/978-3-031-30044-8_9
63. de Vilhena, P.E., Pottier, F., Jourdan, J.: Spy game: verifying a local generic solver in Iris. Proc. ACM Program. Lang. **4**(POPL), 33:1–33:28 (2020). https://doi.org/10.1145/3371101
64. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337
65. Wright, A.K., Jagannathan, S.: Polymorphic Splitting: An Effective Polyvariant Flow Analysis. ACM Trans. Program. Lang. Syst. **20**(1), 166–207 (1998). https://doi.org/10.1145/271510.271523
66. Yi, K.: Compile-time Detection of Uncaught Exceptions in Standard ML Programs. In: Charlier, B.L. (ed.) Static Analysis, First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 28-30, 1994, Proceedings. Lecture Notes in Computer Science, vol. 864, pp. 238–254. Springer (1994). https://doi.org/10.1007/3-540-58485-4_44
67. Yi, K.: An Abstract Interpretation for Estimating Uncaught Exceptions in Standard ML Programs. Sci. Comput. Program. **31**(1), 147–173 (1998). https://doi.org/10.1016/S0167-6423(96)00044-5
68. Yi, K., Ryu, S.: A cost-effective estimation of uncaught exceptions in Standard ML programs. Theoretical Computer Science **277**(1-2), 185–217 (2002). https://doi.org/10.1016/S0304-3975(00)00317-0

Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law

Raphaël Monat[1]*(✉), Aymeric Fromherz[2]*, and Denis Merigoux[2]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Inria Paris, Paris, France
{raphael.monat,aymeric.fromherz,denis.merigoux}@inria.fr

Abstract. Legal expert systems routinely rely on date computations to determine the eligibility of a citizen to social benefits or whether an application has been filed on time. Unfortunately, date arithmetic exhibits many corner cases, which are handled differently from one library to the other, making faithfully transcribing the law into code error-prone, and possibly leading to heavy financial and legal consequences for users.
In this work, we aim to provide a solid foundation for date arithmetic working on days, months and years. We first present a novel, formal semantics for date computations, and formally establish several semantic properties through a mechanization in the F* proof assistant. Building upon this semantics, we then propose a static analysis by abstract interpretation to automatically detect ambiguities in date computations. We finally integrate our approach in the Catala language, a recent domain-specific language for formalizing computational law, and use it to analyze the Catala implementation of the French housing benefits, leading to the discovery of several date-related ambiguities.

Keywords: Verification, Semantics, Abstract Interpretation

1 Introduction

From filesystems to web servers, time representations are pervasive in modern computer systems. While several libraries and standards were proposed throughout the years, current well-established approaches such as Unix time [53] used in the standard C library or Windows' FILETIME [36] represent dates and time as a number of seconds or nanoseconds that have elapsed since an arbitrary date.

This approach is sufficient for many usecases, in particular when dates are only used for logging purposes, or for determining the chronology of two events. However, it does not permit more complex arithmetic, for instance the addition of months or years, that span a variable number of days. For these usecases, mainstream programming languages offer different libraries that adopt different conventions. For example, Python's `datetime` module [46] forbids the addition of months, while Java's `java.time` library [43] silently rounds invalid dates onto the largest pre-existing date, hiding ambiguous computations from programmers.

* R. Monat and A. Fromherz—Equal contribution.

S. Weirich (Ed.): ESOP 2024, LNCS 14577, pp. 421–450, 2024.
https://doi.org/10.1007/978-3-031-57267-8_16

Given the variety of libraries and behaviors across languages, programming with date arithmetic is thus highly error-prone, and developers' assumptions about how dates behave might vary from project to project. When developing systems whose correctness is critical and that heavily depend on date computations, such as expert legal systems that rule our social and financial lives, this issue becomes highly concerning. As an example, consider the following excerpt from Section 121 of the US Internal Revenue Code [25], which defines the "Exclusion of gain from sale of principal residence".

> In the case of a sale or exchange of property by an unmarried individual whose spouse is deceased on the date of such sale, paragraph (1) shall be applied by substituting "$500,000" for "$250,000" if such sale occurs not later than 2 years after the date of death of such spouse and the requirements of paragraph (2)(A) were met immediately before such date of death.

This paragraph differentiates between two cases, depending on whether a sale occurred *not later than 2 years* after a given date. While applying this paragraph is straightforward in most real-world cases, corner cases raise interesting questions. In particular, when considering leap years, what should be the result of adding two years to February 29th? When manually computing taxes, lawyers would be able to detect the ambiguity, and to reach a decision based on legal precedents. If handled automatically by a computer however, the computation may be done incorrectly; computing February 29 2004 + 2 years in Java using `java.time` would return February 28 2006, while performing the same computation using the `date` utility from Coreutils returns March 1 2006.

Similar computations are pervasive in expert legal systems; the corresponding regulations rely on them to determine whether a citizen is eligible to social benefits or a resident for tax purposes. Errors in such systems can have dramatic consequences; case in point, the incorrect implementation of Louvois, the former French military payroll system, led to several families either receiving over-payments that they had to reimburse years later, or incomplete paychecks totaling a few cents [42]. For such critical software, it is therefore paramount to provide clear semantics for date computations to avoid mistakes based on erroneous assumptions about a library's behavior. Additionally, such a semantics can form the basis for further analyses, paving the way for the automated detection of date-related ambiguities as part of the development process.

Unfortunately, while elegant in theory, a universal semantics for dates and date arithmetic would not be usable in practice; when possible ambiguities are identified in law texts, legislators oftentimes extend or modify the law itself to avoid them. For instance, article 641 of the French civil procedure code [30] specifies that, when adding a positive duration to a date to compute a deadline, the rounding, if needed, should go down. Such articles often have narrow application scopes; similar articles in other branches of the law might either leave rounding unspecified, or adopt a different convention. In the US, date computations when filing motions are heavily specified, however the complexity and amount of corner cases led to no less than 27 subsequent notes and amendments to provide clarifications [14]. Other regulations instead attempt to escape ambiguities due to month or year additions by reducing such computations to a nonambiguous number of days. Such regulations heavily vary depending on the country and the branch of law considered: acts from the Council of European Communities consider that a month should be treated as 30 days [15], while the Indian Supreme Court took the opposite approach, enacting that the duration of a month for customs purposes is variable [4]. To enable their adoption in a variety of contexts, date libraries therefore require their semantics to be configurable by developers.

The lowest granularity of date arithmetic we focus on is the day level. Our literature review and communications with lawyers in different countries have indeed shown that this kind of date arithmetic is sufficient for the kind of tax and social benefits computations that are the core application target of Catala.

In this paper, we aim to provide a sound foundation for critical software relying on date computations, through the following contributions:

Formally Capturing Date Computations. We first present a formal semantics of date computations (Sec. 2). Our formalization relies on a base semantics, which is universal and does not specify a rounding mode but instead provides facilities to round on-demand. We leverage these facilities to derive a rounding-specific semantics for different rounding policies. We mechanize this semantics in the F* proof assistant, and prove several theorems establishing necessary conditions for, e.g., the monotonicity or associativity of computations (Sec. 3). As part of this mechanization, we also identify seemingly intuitive properties that do not hold in practice, and exhibit counter-examples.

Automatically Detecting Date Ambiguities. Building on the semantics, we define a notion of *rounding-insensitivity*, which captures that the result of evaluating a program's expression does not depend on the chosen rounding policy (Sec. 4). Aiming to automatically identify possibly harmful ambiguities, we then propose a new static analysis based on abstract interpretation [16] targeting this 2-safety hyperproperty. We implement our analysis in the Mopsa static analyzer [28, 29]. We show that with relational numerical abstract domains, our analysis enables precise reasoning. In addition, our implementation provides actionable counter-example hints which will help users understand why a given expression is rounding-sensitive.

Contribution to Date Arithmetic Libraries. To enable the adoption of this work in existing projects, we implement an OCaml library abiding by our formal semantics, which exposes common rounding modes, as well as an option to abort when ambiguous computations are detected. Our library is standalone and open-source, and easily integrable in OCaml developments. We also survey the behavior of mainstream date arithmetic libraries (Sec. 6), and provide litmus tests that can be used to easily understand how a library behaves with respect to date rounding.

Case Study: Integration in the Catala Language. To demonstrate the applicability of our approach in real-world programs, we replace previous han-

date unit	$\delta ::= y \mid m \mid d$
rounding mode	$r ::= \uparrow \mid \downarrow \mid \perp$
values	$v ::= (y, m, d) \mid \perp$
expressions	$e ::= v \mid e +_\delta n \mid \text{rnd}_r\ e$
period	$p ::= (n_d, n_m, n_y)$

Fig. 1. Date expressions

dling of dates in the Catala language [34], a recent domain-specific language for formalizing computational law, by our library. We also extend the Mopsa [28, 29] static analyzer to support a subset of the Catala language, enabling us to analyze Catala programs for rounding-insensitivity. We evaluate our approach against an existing Catala implementation of the French housing benefits, and automatically identify several date-related ambiguities in the Catala model. This work is in the process of being upstreamed in the Catala compiler.

2 Formalizing Date Arithmetic

We start this section by presenting a base semantics for date computations, which does not explicitly specify a rounding policy to handle ambiguous dates. Dates expressions are presented in Fig. 1. Dates values are represented in the year-month-day format of the standard Gregorian calendar, where each component will be represented as an integer. We also include a $\perp$ element, which represents an error case. Date expressions consist of either date values, or of the application of one of the date operators. Date expressions also contain variables, however their treatment is straightforward and orthogonal to this work; we omit them as well as their associated environment in our presentation. Operators are of two kinds: the addition $+_\delta$ of n years, months, or days, where n is an integer, and the rounding rnd_r of a date. Our semantics supports three types of rounding: $\text{rnd}_\uparrow$ rounds up the current date to the nearest valid date; $\text{rnd}_\downarrow$ rounds down, and $\text{rnd}_\perp$ raises an error if the current date is invalid. A period is a triple of relative integers, respectively representing the numbers of days, months and years.

We now define a formal semantics for evaluating expressions. We start by describing the semantics of date addition, presented in Fig. 2. To match standard date formats, we start counting at 1 for valid days and months; to simplify the presentation, we will often represent months using their name instead of their number (e.g., `Jan` instead of 1). Our semantics is designed to preserve the following invariant: assuming the date on the left is initially valid, any non-ambiguous computation will return a valid date. When the computation is ambiguous, the resulting date is between the largest smaller and the smallest larger valid date.

Our semantics is defined recursively. Consider for instance the addition of a number of days n. If n is small enough to remain in the same month and year, we are in the terminal case and the rule Add-Days applies. The first premise of the rule ensures that the date is initially valid. It relies on an auxiliary function nb_days, omitted for brevity, which computes the number of days for a month in a given year (e.g., 31 for January, and 28 or 29 for February depending

$$\textsc{Add-Year}\quad \frac{}{(y, m, d) +_y n \to (y + n, m, d)} \qquad \textsc{Add-Month-Under}\quad \frac{m + n < 1}{(y, m, d) +_m n \to (y - 1, m, d) +_m (n + 12)}$$

$$\textsc{Add-Month}\quad \frac{1 \leq m + n \leq 12}{(y, m, d) +_m n \to (y, m + n, d)} \qquad \textsc{Add-Month-Over}\quad \frac{m + n > 12}{(y, m, d) +_m n \to (y + 1, m, d) +_m (n - 12)}$$

$$\textsc{Add-Days-Over}\quad \frac{1 \leq d \leq \text{nb\_days}(y, m) \qquad d + n > \text{nb\_days}(y, m)}{(y, m, d) +_d n \to ((y, m, 1) +_m 1) +_d (n - (\text{nb\_days}(y, m) - d) - 1)}$$

$$\textsc{Add-Comp}\quad \frac{e \to e'}{e +_\delta n \to e' +_\delta n} \qquad \textsc{Add-Days-Under1}\quad \frac{1 < d \leq \text{nb\_days}(y, m) \qquad d + n \leq 0}{(y, m, d) +_d n \to (y, m, 1) +_d (d - 1 + n)}$$

$$\textsc{Add-Days-Err1}\quad \frac{d < 1}{(y, m, d) +_d n \to \bot} \qquad \textsc{Add-Days-Under2}\quad \frac{n + 1 \leq 0 \qquad (y, m, 1) +_m (-1) \to (y', m', d')}{(y, m, 1) +_d n \to (y', m', 1) +_d (n + \text{nb\_days}(y', m'))}$$

$$\textsc{Add-Days-Err2}\quad \frac{d > \text{nb\_days}(y, m)}{(y, m, d) +_d n \to \bot} \qquad \textsc{Add-Days}\quad \frac{1 \leq d \leq \text{nb\_days}(y, m) \qquad 1 \leq d + n \leq \text{nb\_days}(y, m)}{(y, m, d) +_d n \to (y, m, d + n)}$$

Fig. 2. Semantics for date addition

on the year). Otherwise, we either add a month (rule Add-Days-Over) or remove a month (rule Add-Days-Under2) and perform a new addition with an updated number of days. When the initial date is invalid, we return $\bot$ to avoid propagating large errors and maintain important properties about date semantics that we prove in Sec. 3. When composing additions, it might therefore be necessary to apply rounding operators presented later in this section to avoid $\bot$. One last point of interest in these semantics is the dissymmetry between the Add-Days-Over and Add-Days-Under-* rules. Since adding a number of days is never ambiguous, we wish to ensure that, assuming the initial date is valid, we never apply the Add-Days-Err1 or Add-Days-Err2 rules. To do so, when updating the month or year during day addition, we always go through an intermediate state corresponding to the first day of the month, which is always a valid day independently of the month and year. For brevity, we also omit several redundant error cases, where the current month does not belong to the interval $[1; 12]$; these cases return $\bot$. Following standard notations, we will denote the transitive closure of our small-step semantics as $\overset{*}{\to}$.

The last step is now to define semantics for rounding, shown in Fig. 3. Compared to additions, the rounding semantics is simpler: if the date is already valid, any mode of rounding leaves the date unchanged (Round-Noop). Otherwise, rounding down (Round-Down) returns the last day of the current month, rounding up (Round-Up) returns the first day of the next month, while the

$$\textsc{Round-Err1}\ \frac{d < 1}{\text{rnd}_r(y,m,d) \to \bot} \qquad \textsc{Round-Err2}\ \frac{d > \text{nb\_days}(y,m)}{\text{rnd}_\bot(y,m,d) \to \bot} \qquad \textsc{Round-Down}\ \frac{d > \text{nb\_days}(y,m)}{\text{rnd}_\downarrow(y,m,d) \to (y,m,\text{nb\_days}(y,m))}$$

$$\textsc{Round-Noop}\ \frac{1 \leq d \leq \text{nb\_days}(y,m)}{\text{rnd}_r(y,m,d) \to (y,m,d)} \qquad \textsc{Round-Up}\ \frac{d > \text{nb\_days}(y,m) \qquad (y,m,d) +_m 1 \xrightarrow{*} (y',m',d')}{\text{rnd}_\uparrow(y,m,d) \to (y',m',1)}$$

Fig. 3. Semantics for date rounding

strict rounding mode (Round-Err2) raises an error. In all cases, if the day is initially negative, rounding returns $\bot$; we will prove in Sec. 3 that this never happens when starting from a valid date.

Separating additions and rounding has several benefits. Different use cases might require different rounding modes, and different ways of adding days, months, and years. For instance, when adding a period such as 1 year and 10 months, some settings might specify that months should be added first, or that rounding must be performed after adding months, and again after adding years; our formal semantics enables this flexibility.

The last remaining step is to define additions not just for individual days, months, or years, but for composite time periods. Building upon our semantics, we can define this generically for a rounding mode r as follows, and avoid the need for users to manually call rounding operators.

$$e +_r (y,m,d) ::= \text{rnd}_r(((e +_y y) +_m m)) +_d d$$

One point of interest in our derived forms is that we only apply rounding after performing addition of years and months. Indeed, adding a year should be equivalent to adding 12 months. However, if we performed rounding after each operation, adding 1 year and 1 month to February 29 2020 with the rounding-up mode would return April 1, 2021 instead of Mar 29, 2021. We emphasize that, in cases where this behavior would be expected, defining derived forms corresponding to this semantics would be straightforward using our base semantics.

Based on this semantics, we can now formally define the notion of an ambiguous date expression in Definition 1.

Definition 1 (Ambiguous expression). *A date expression e is ambiguous if and only if $rnd_\bot(e) \xrightarrow{*} \bot$.*

Note that this intensional definition of ambiguity is equivalent to stating that the an expression e is ambiguous if and only if rounding e in different modes yields different dates.

While the semantics presented in this section focuses on the core, possibly ambiguous computations, our work also includes other non-ambiguous operators (omitted for brevity), e.g., to retrieve the first or last day of a given month. This allows to encode a variety of patterns, for instance, the second-to-last day of a month by combining date arithmetic with the "last day of month" operator, or to rely on a preprocessing phase if months must be treated as 30 days [15]. Our semantics supports reasoning on computations mixing rounding modes.

3 Mechanizing Semantics

Building upon the semantics presented in the previous section, we now present several properties of interest related to date computations that we will rely upon when designing a static analysis in Sec. 4. As part of our contributions, we mechanize our semantics, related properties and their proofs inside the F* proof assistant [52].

3.1 Semantic properties

As part of our proof development, we separate semantic properties in two categories: properties established on the base semantics, valid for all derived forms, and properties derived on specific rounding modes. In many cases, proofs on derived forms can be performed efficiently by composing lemmas on base semantics, thus simplifying the proof effort. During development, we also encode our OCaml implementation of date computations and corresponding theorems into qcheck [54], a QuickCheck [13] inspired property-based testing framework for OCaml. We mostly used QuickCheck as a fast sanity check before spending time proving lemmas in F*. In particular, our initial intuition for several of the lemmas and theorems presented was often unreliable, omitting corner cases; we used QuickCheck to gain more confidence in our intuition before moving to F*. This encoding allowed us to automatically find most of the counter-examples presented in Sec. 3.2.

We start by proving that expressions in our semantics always evaluate to a value (possibly $\bot$), i.e., reduction is never stuck and it terminates.

Theorem 1 (Normalization). *For any date d, and any integer n, there exists a value v_δ such that $d +_\delta n \xrightarrow{*} v_\delta$.*

In addition to normalization, a useful property about our semantics is a characterization of valid computations: when using any of the non-abort rounding modes, an addition starting from a valid date will always return a valid date; the definition of validity is straightforward, but omitted for brevity. To prove it, we need the following properties on base semantics, which we prove by induction on the reductions.

Lemma 1 (Well-formedness of day addition). *For any valid date d, any integer n, and any value v, $d +_d n \xrightarrow{*} v \Rightarrow v \neq \bot$.*

Lemma 2 (Well-formedness of year/month addition). *For any valid date d, any integer n, any value v, and $\delta \in \{y, m\}$, we have $d +_\delta n \xrightarrow{*} v \Rightarrow v \neq \bot \land \mathrm{day\_of}(v) \geq 1$.*

Lemma 3 (Well-formedness of rounding). *For any date d such that $d \neq \bot$, any value v, and $r \in \{\uparrow, \downarrow\}$, we have $rnd_r\ d \overset{*}{\to} v \Rightarrow valid(v)$.*

We can now state the following theorem on the derived semantics.

Theorem 2 (Well-formedness). *For any valid date d, any period p, any value v, and $r \in \{\downarrow, \uparrow\}$, we have $d +_r p \overset{*}{\to} v \Rightarrow valid(v)$.*

We now present several theorems related to the monotonicity of the addition in our semantics. Date comparison is defined in the standard way, as the lexicographical order on (y, m, d). To simplify the presentation, we lift the comparison operators to operate on date expressions, defined as the comparison on the values obtained by evaluating the expressions.

Theorem 3 (Monotonicity). *For any dates d_1, d_2, for any period p, for $r \in \{\downarrow, \uparrow\}$, if $d_1 < d_2$, then $d_1 +_r p \leq d_2 +_r p$.*

A point of interest in this theorem is the discrepancy between bounds: while the bound in the premise is strict, the bound in the conclusion is loose. Unfortunately, a stronger version with strict bounds on both sides does not hold; for instance, two additions involving rounding down of April 30 and April 31 respectively yield the same result. To prove this theorem, we again need several intermediate lemmas operating on base semantics. First, we establish an equivalence between adding years and adding months. We then state and prove several monotonicity properties on the base semantics. The proof of Theorem 3 follows by direct application of these lemmas.

Lemma 4 (Equivalence of year and month addition). *For all date d, for all integer n, $d +_y n = d +_m (12 * n)$.*

Lemma 5 (Monotonicity of year and month addition). *For all dates d_1, d_2, for any integer n, for $\delta \in \{y, m\}$, $d_1 < d_2 \Rightarrow d_1 +_\delta n < d_2 +_\delta n$.*

Lemma 6 (Monotonicity of day addition). *For all valid dates d_1, d_2, for any integer n, $d_1 < d_2 \Rightarrow d_1 +_d n < d_2 +_d n$.*

Lemma 7 (Monotonicity of rounding). *For all dates d_1, d_2, for $r \in \{\downarrow, \uparrow\}$, $d_1 < d_2 \Rightarrow rnd_r(d_1) \leq rnd_r(d_2)$.*

Finally, we state the following lemma, which guarantees that rounding down will always return a smaller date than rounding up. Additionally, when the addition is not ambiguous, the two rounding modes return the same result.

Theorem 4 (Rounding).

1. *For all date d, for all period p, $d +_\downarrow p \leq d +_\uparrow p$.*
2. *For all date d, for all period p, $d +_\bot p \neq \bot \Rightarrow d +_\downarrow p = d +_\uparrow p = d +_\bot p$.*

We finally characterize the ambiguity of month addition, a property that we will need to prove the soundness of the static analysis presented in Sec. 4.

Theorem 5 (Characterization of ambiguous month additions). *For all valid date d, for all integer n, for all value v such that $d +_m n \overset{*}{\to} v$, we have* $\text{nb\_days}(\text{year\_of}(v), \text{month\_of}(v)) < \text{day\_of}(v) \Leftrightarrow rnd_\bot(v) \overset{*}{\to} \bot$.

3.2 Non-properties and counter-examples

We now present several seemingly intuitive and ideally useful properties about date semantics that do not hold in practice.

Non-Property 1 (Commutativity of addition) *For all date d, for all periods p_1, p_2, for all $r \in \{\downarrow, \uparrow\}$, we have $(d +_r p_1) +_r p_2 = (d +_r p_2) +_r p_1$*

Consider the case where $d =$ March 31, $p_1 = 1$ day, and $p_2 = 1$ month. When adding p_1 first and rounding down, the addition returns April 30, while the result when adding p_2 first will be May 1. Similar examples exist when rounding up, for instance, by setting $d =$ January 29 2023 , $p_1 = 30$ days, and $p_2 = 1$ month.

Non-Property 2 (Associativity of addition) *For all date d, for all periods p_1, p_2, for $r \in \{\downarrow, \uparrow\}$, we have $(d +_r p_1) +_r p_2 = d +_r (p_1 + p_2)$*

Consider the case where $d =$ March 31, $p_1 = 1$ month, and $p_2 = 1$ month. In all rounding modes, adding p_1 followed by p_2 will require rounding, ultimately yielding May 30 or June 1, while directly adding $p_1 + p_2$ returns May 31.

As the addition being associative and commutative is common among most datatypes, we emphasize that its invalidity for dates can be a source of confusion for programmers; common optimizations or rewritings of date computations in a seemingly equivalent way (e.g., replacing 1 month + 1 month by 2 months) can lead to different outcomes. However, these disparities are exclusively due to occurrences of rounding in computations. We thus aim to help programmers when handling date computations by proposing a static analysis that automatically detects when rounding might impact the evaluation of expressions.

4 A Static Analysis For Rounding-Insensitivity

In this section, we leverage our formal semantics to define a sound static analysis automatically verifying date computations programs. Our goal is to statically detect ambiguous computations, whose result depends on the chosen rounding mode. Indeed, when writing programs whose specification is the law, choosing the rounding mode arbitrarily is not a possibility; this would amount to a legal interpretation that exposes the administration operating the program to be challenged in court if the rounding mode is unfavorable to a user. The cost of bearing the responsibility for making technical regulatory choices for administration personnel has been documented by Torny [55].

A naive approach would be to flag any program which contains an ambiguous addition. However, this solution can be overly restrictive: computations can be ambiguous while having no impact on the final outcome of the program. Consider for example the expression `d + 1 month <= March 15 2023`. If no rounding happens when adding `d` and `1 month`, then the expression is obviously safe. Otherwise, we notice that the rounding may only happen to yield the last day of a month, or the next day of the upcoming month. In both cases, comparing

```
1  date current = random_date();
2  date birthday = random_date();
3  date intermediate = birthday + [2 years, 0 months, 0 days];
4  date limit = first_day_of(intermediate);
5  assert(sync(current < limit));
```

Fig. 4. Example extracted from Catala code modeling the French housing benefits

this result with a date in the middle of a month is thus safe. Instead, we consider a more interesting property called *rounding-insensitivity*, capturing that the evaluation of an expression is the same for both rounding modes.

At a high-level, our analysis works by tracking constraints over the day, month, and year of a date, through the YMD domain (Sec. 4.1). The YMD domain is fully parametric in a numerical abstract domain, and works by translating date constraints into numerical constraints. We discuss the choice of numerical abstract domains in Sec. 4.2, in order to obtain the best precision in the presence of linear constraints and unconstrained dates. We analyze the computations with both rounding modes and compare the result to decide rounding-insensitivity, which is a 2-safety hyperproperty. We explain how we lift the YMD domain to these double computations in Sec. 4.3. We implemented our analysis within the Mopsa static analysis platform [28, 29], described in Sec. 4.4. We have taken special care in ensuring that actionable counter-examples can be generated in Sec. 4.5, paving the way for use by non-experts.

We think that abstract interpretation hits a sweet spot to perform this analysis. Its full automation makes it usable by non-specialists, especially with the provided counter-example hints. It allows to derive tailored approximations thanks to Th. 5. The current definition of date addition is recursive and there are non-linear arithmetic constraints involved, which does not work well with SMT.

We use as a motivating example the program shown in Fig. 4. This program has been extracted from a Catala code snippet used to formalize the French housing benefits [33, Sec. 3.1]. We will provide more details on Catala and the extraction to date programs in Sec. 5. In this program, we pick two arbitrary, unconstrained dates, perform a date-duration addition of two years, and project the resulting date onto the first day of its month. The assertion at line 5 expresses the rounding-insensitivity of the comparison between an arbitrary, unconstrained date and the computed date.[3] The `sync` predicate, formally defined in Sec. 4.3, holds if and only if the evaluation of its expression in both rounding modes yields the same result, meaning that the expression is rounding-insensitive.

The programs we consider in this section are written in a standard, toy imperative language.

4.1 The YMD domain combinator

The YMD domain translates constraints on the year, month and day of a date into numerical constraints over three integer variables. These numerical constraints are handled by a numerical abstract domain, described in Definition 2.

[3] Here `sync(current < limit)` could be reduced to `sync(limit)`. However our analysis will not need it, and will be able to provide counter-example hints also targeting the values of `current`, improving readability of the output.

$$\text{dates\_dom} : \begin{cases} (\mathcal{V} \to \mathbb{Z}) \to \mathcal{P}(\mathcal{V}) \\ \quad\rho \quad\mapsto \{v \mid \text{year}(v), \text{month}(v), \text{day}(v) \in \text{dom}(\rho)\} \end{cases}$$

$$\gamma_{\text{YMD}} : \begin{cases} \mathcal{N}^{\#} \to \mathcal{P}(\mathcal{V} \to \mathcal{D}) \\ n^{\#} \mapsto \bigcup_{\rho \in \gamma_{\mathcal{N}}(n^{\#})} \{e \mid \text{dom}(e) = \text{dates\_dom}(\rho) \land \forall v \in \text{dom}(e), e(v) = (y, m, d) \\ \qquad \land \text{valid}(y, m, d) \land y = \rho(\text{year}(v)) \land m = \rho(\text{month}(v)) \land d = \rho(\text{day}(v))\} \end{cases}$$

Fig. 5. Concretization of the YMD domain

The YMD domain can be seen as a domain combinator, or a functor relying on a numerical abstract domain – we will discuss the chosen instantiation in Sec. 4.2. This domain works at a fixed rounding mode.

Definition 2 (Numerical abstract domain). *In the following, a numerical abstract domain is a lattice $\mathcal{N}^{\#}$ on which the following operations are defined:*
- *The assignment,* `assign`*, between a variable and an expression in a given abstract environment yields another abstract environment.*
- *The boolean filtering of a state,* `assume`*, filters an abstract environment to enforce that a boolean expression holds.*

This domain is further defined by a concretization function $\gamma_N : \mathcal{N}^{\#} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$ mapping numerical abstract environments to a set of concrete integer environments it represents. We assume the numerical abstract domain is sound.

Given a date variable v, the YMD domain will create new auxiliary (or ghost) variables $\text{year}(v), \text{month}(v), \text{day}(v)$, which do not exist in the original program but simplify reasoning. This is an approach we borrow from the deductive verification community, and that has been used in static analyses both in the work of Chevalier and Feret [12] as well as in Mopsa.

We provide a formal definition of the concretization, which defines the meaning of the YMD domain, and illustrate it on an example.

Definition 3 (Concretization of the YMD domain). *The concretization of the YMD domain is formally defined in Fig. 5. It explains how an abstract numerical environment $n^{\#} \in \mathcal{N}^{\#}$ can be interpreted into a set of date environments $e \in \mathcal{V} \to \mathcal{D}$ mapping variables to dates. To construct these date environments, we first pick an integer environment $\rho \in \mathcal{V} \to \mathbb{Z}$ from the concretization of the numerical abstract domain $\gamma_N(n^{\#})$. The date environments will have as domain definition the date domain of function ρ,* $\text{dates\_dom}(\rho)$*, which is the set of variables where auxiliary year, month and day variables are defined in ρ. For each of those variables $v \in \text{dates\_dom}(\rho)$, $e(v)$ corresponds to the date defined by the auxiliary variables in ρ, provided that the date is valid.*

Example 1 (Concretization). Let us assume our numerical domain is a map from variables to intervals, and consists of the following state: $n^{\#} = \text{day}(d) \in [1, 31] \land \text{month}(d) \in [1, 12] \land \text{year}(d) = 2023$. In that case, the concretization is the set of date environments e defined on variable d such that $e(d)$ can be any valid date of 2023. Thus, there is a date environment $e \in \gamma_{\mathrm{YMD}}(n^{\#})$ such that $e(d) = (2023, 1, 31)$. However, there is no date environment such that $e(d) = (2023, 2, 29)$ and $e \in \gamma_{\mathrm{YMD}}(n^{\#})$ because the date is invalid (2023 is not a leap year).

The YMD domain handles the following transfer functions:

– Accessors to the day, month or year number of a date. Given a date encoded as a variable v, these functions return the associated variable $\mathrm{day}(v), \mathrm{month}(v)$, $\mathrm{year}(v)$ respectively.
– Projection of a date on the first day of the month: given a date encoded as a variable v, this function creates a new date having the same auxiliary month and year variables. The day auxiliary variable is set to 1. A similar operator working on the last day of the month can be defined.
– The main part of the YMD domain is the transfer function handling month addition and potential rounding originating from this addition. We define it below, argue it is sound, and illustrate it on an example (in Sec. 4.2). As we have proved in Lemma 4, additions on years and months can be reduced to additions on months. Our current, potentially ambiguous, real-world examples taken from legislative code do not use day addition; as it is never ambiguous, we thus do not currently implement it. Given its similarity to month addition, we do not foresee any technical difficulty doing so.
– The YMD domain also provides a transfer function to compare two dates. It is induced by the lexicographic definition of concrete date comparisons and partitions the results to improve the precision.

Transfer function for month addition. We provide a simplified OCaml implementation for the month addition transfer function in Fig. 6. The transfer function takes as parameter a `date`, represented as a variable; a concrete number of months; an input abstract state; and a rounding mode chosen for date computations. It will return a case disjunction[4] of type `cases`: a list of `case`, each consisting in an expression and an abstract state. We start by defining `day, month, year`, which are expressions representing the day, month and year number of `date` through auxiliary variables. The resulting month and year are computed through non-linear expressions. Similarly to the semantics, we encode months as integers to perform arithmetic operations, and start our numbering at 1 for January. The transfer function performs a case disjunction to detect if date rounding will happen, following the characterization of ambiguous month additions (Th. 5). This case disjunction checks whether the day of the date is compatible with the number of days in the resulting month (and year, as February has one more day during leap years). This disjunction is encoded thanks to the `switch` utility, which takes as input an abstract state and a list of tuple of expressions and continuations. Given a tuple `(cond, k)`, the input abstract state is filtered to satisfy the expression `cond` (by delegation to the numerical

[4] These disjunctions can be seen as a partitioning of the abstract state. In this section we consider everything is partitioned to improve the precision. Our implementation supports limiting the number of partitions.

abstract domain). The resulting abstract state is fed to the continuation, which yields a case. The cases we encounter during the addition are:

Rounding to 29 Feb. of a leap year. If the resulting month is February of a leap year, and the current day number is greater than 29, we will have to perform date rounding. We do so using the auxiliary `round` function. Depending on the rounding mode, it either chooses the provided date, or the first of the month afterwards. This date is then returned in its corresponding abstract state using `mk_date`, whose implementation is not detailed.

Rounding to 28 Feb. of a non-leap year. Similar case omitted for brevity.

Rounding to a 30-day month. If the current day number is 31 but the resulting month has 30 days (i.e, it is either April, June, September or November), we also have to perform a rounding, either to the 30th of the resulting month, or the 1st of the month after.

Other cases. No rounding happens, the day number remains the same.

Note that `add_months`, `round` and `is_leap` define syntactic expressions, which will be delegated through `assign` and `assume` to the numerical abstract domain. The expressions at lines 6, 13, 14, 21–22, 26, 28, 30 are not directly evaluated: they will be interpreted by the `assume` of the numerical abstract domain during the evaluation of the `switch` function. The definition of the transfer function for month addition assumes the number of months to add is known as a concrete integer. This is not restrictive in practice: all programs we extracted from Catala in Sec. 5 only perform date-month addition with a concrete number of months.

The proof of soundness of the abstract month addition, is not formalized in F* and omitted for brevity. However, it is a direct application of the characterization of ambiguous month additions established in Th. 5, and proved formally in F*.

The analysis may refine constraints on a day, month or year auxiliary variable. These constraints could then entail new constraints on other auxiliary variables of the same date to represent only valid dates. This propagation phase is performed by the strengthening operator described below, which is sound as it only removes invalid dates, which are not taken into account by the concretization.

Strengthening operator. The strengthening operator enforces the following:
- If the month is February, the day is less than 30.
- If the month is April, June, September of November, the day is less than 31.
- If the date is February 29, we know the current year is a leap year. We enforce that the year number is divisible by 4, which is a necessary condition.

Comparison transfer function. The transfer function for date comparisons is `dates_lt` in Fig. 6; it encodes a lexicographic comparison.

4.2 Instantiating YMD with a combination of numerical domains

The YMD domain is fully generic in the numerical abstract domain it relies on to translate date constraints into constraints over integers. We describe how we

```
1   type case = expr * state
2   type cases = case list
3
4   let switch abs = List.map (fun (cond : expr, k : state -> case) -> k (assume cond abs))
5
6   let is_leap (y : expr) : expr = (y % 4 = 0 && y % 100 <> 0) || (y % 400 = 0)
7
8   let round (r : rounding) (d m y : expr) (abs : state) : case =
9     match r with
10    | RoundDown ->
11      mk_date d m y abs
12    | RoundUp ->
13      let succ_m = 1 + res_month % 12 in
14      let succ_y = y + res_month / 12 in
15      mk_date 1 succ_m succ_y abs
16
17  let add_months (r : rounding) (date : var) (nb_m : int) (abs : state) : cases =
18    let day = day_of date in
19    let month = month_of date in
20    let year = year_of date in
21    let res_month = 1 + (month - 1 + nb_m) % 12 in
22    let res_year = year + (month - 1 + nb_m) / 12 in
23    switch abs
24    [
25      (* Rounding to 29 Feb. of a leap year *)
26      day > 29 && res_month = Feb && is_leap res_year, round r 29 res_month res_year;
27      (* Rounding to 28 Feb. of a non-leap year *)
28      day > 28 && res_month = Feb && not (is_leap res_year), round r 28 res_month res_year;
29      (* Rounding to a 30-day month *)
30      day > 30 && is_one_of res_month [Apr;Jun;Sep;Nov], round r 30 res_month res_year;
31      (* No rounding *)
32      mk_true, mk_date day res_month res_year
33    ]
34
35  let dates_lt (d1 d2 : var) (abs : state) : cases =
36    switch abs
37    [
38      (year_of d1) < (year_of d2), mk_true;
39      (year_of d1) > (year_of d2), mk_false;
40      (year_of d1) = (year_of d2) && (month_of d1 < month_of d2), mk_true;
41      (year_of d1) = (year_of d2) && (month_of d1 > month_of d2), mk_false;
42      (year_of d1) = (year_of d2) && (month_of d1 = month_of d2)
43        && (day_of d1 < day_of d2), mk_true;
44      (year_of d1) = (year_of d2) && (month_of d1 = month_of d2)
45        && (day_of d1 >= day_of d2), mk_false;
46    ]
```

Fig. 6. Abstract transfer functions for month addition and date comparison

chose a combination of numerical abstract domains to get the best precision possible in the presence of non-linearity and unconstrained dates.

We initially started using intervals and congruences for our first tests. Due to its convexity, the interval domain was unable to precisely represent months where the day number may be rounded to 30 days during the date-month addition (line 30 of Fig. 6). Thus, we added a domain of powerset of integers (of size at most 4) to be precise enough for this usecase. When `month` is not a constant, the congruence domain will be unable to precisely represent the resulting month (line 21 of Fig. 6), and refine the potential values of `month` given constraints on `res_month`. This situation happens often in our evaluation; it is shown in our motivating example. We resolved this precision issue by switching from the congruence domain to the relational, linear congruence domain [5]. We also added the polyhedra domain [17] to keep track of equalities between different day, month and year variables, which happens during analyses on programs with unconstrained dates, as we will show in the upcoming examples.

Our current numerical abstract domain is a reduced product between grids, polyhedra, intervals, and a bounded powerset of integers. The relational domains rely on the Apron library [27]. The approximation of non-linear computations is performed through linearization techniques [37].

Example 2. Let us consider the program below picking an arbitrary, unconstrained date `d` and then adding one month to `d`. We illustrate the different cases of the transfer function `add_months` in this case, assuming we round down.

```
date d = random_date(); date d2 = d + [0 years, 1 months, 0 days];
```

Rounding to 29 Feb. of a leap year. In the first case of the transfer function, the numerical domain is able to deduce from the expression `day > 29 && res_month = Feb` that the day of `d` is either 30 or 31, and the month is January. In the rounding down mode, `d2` is thus February 29th. The relational domain additionally expresses that $\text{year}(d) = \text{year}(d2)$.
Rounding to 28 Feb. of a non-leap year. Similar case, omitted for brevity.
Rounding to a 30-day month. The numerical abstract domain infers that `d` represents the 31st of March, May, August or October, tracked thanks to the bounded set of integers domain. As we round down, we deduce that the day of `d2` is 30, and $\text{month}(\texttt{d}) \in \{\texttt{Apr}, \texttt{Jun}, \texttt{Sep}, \texttt{Nov}\}$. In that case, the relational domain can also infer that $\text{year}(d) = \text{year}(d2)$, as `m / 12` will always be zero.
Other cases. In the last case, the intervals and powerset domains cannot express interesting constraints on `d` and `d2`. The relational domains are however able to capture key relations:

- The day does not change as there is no rounding: $\text{day}(\texttt{d}) = \text{day}(\texttt{d2})$.
- Thanks to the grids domain [5] we can infer linear relations modulo a constant, and thus that the month of `d2` is the month after `d`, even if the year changes: $\text{month}(\texttt{d2}) \equiv_{12} \text{month}(\texttt{d}) + 1$, where $\equiv_{12}$ denotes congruence modulo 12. Note that since $\text{month}(\texttt{d2})$ is not a constant, the non-relational congruence domain is not sufficient to express this relation.
- The year number may be the same, or the successor provided that the month of `d` is December. We lose a bit of precision, as the last month always creates a year increase in the concrete.
  $12\,\text{year}(\texttt{d}) + \text{month}(\texttt{d}) \leq 12\,\text{year}(\texttt{d2}) + 11 \wedge 12\,\text{year}(\texttt{d2}) \leq 12\,\text{year}(\texttt{d}) + \text{month}(\texttt{d}) + 1$

Example 3 (Addition and strengthening). We use our running example from Fig. 4, and show what the date addition and the strengthening operator yield for dates `birthday` and `intermediate`. In this example, we assume the dates are rounded up. As we add two years to `birthday`, two of the four cases described in the month addition previously presented will not apply; we omit them below.
Rounding to 28 Feb. of a non-leap year. In that case, `birthday` is a Feb. 29th, and `intermediate` rounds up to March 1st. We additionally know that $\text{year}(\texttt{birthday}) + 2 = \text{year}(\texttt{intermediate})$. The strengthening ensures that $\text{year}(\texttt{birthday})$ is divisible by 4.

No rounding. The day and month numbers of `birthday` and `intermediate` are equal. The year condition is similar to the one provided in Ex. 2.

Example 4 (Comparison). Let us continue with our running example, assuming we are focusing on the partition where `intermediate` has been rounded up to March 1st (as shown in Ex. 3). In that case, `limit` is equal to `intermediate`. Assuming the comparison `current < limit` holds, we have three different cases, described by the line number in Fig. 6. Line 38 yields year(`current`) < year(`limit`). Line 40 enforces year(`current`) = year(`limit`), month(`current`) < month(`limit`), so month(`current`) $\in$ {`Jan`, `Feb`}. Line 42 yields that the year and month numbers of `current` and `limit` are the same and day(`current`) < day(`limit`). This last case is impossible given that $1 \leq$ day(`current`) ≤ 31 and day(`limit`) $= 1$.

4.3 Lifting to both rounding modes

The YMD domain operates at a given, fixed rounding mode. In this section, we leverage the YMD domain to perform date computations in both rounding modes and thus prove rounding-insensitivity. This lifting is inspired by Delmas et al. [21], who analyze product programs to prove endianness portability of C programs. Here, we keep the product of programs implicit: only the rounding mode changes between the two executions we will consider.

We start by explaining how the concrete semantics are lifted from a single rounding mode to both. We assume we have a semantics of expressions (respectively statements) $\mathbb{E}_r[\![expr]\!]$ (resp. $\mathbb{S}_r[\![stmt]\!]$) parameterized by a date rounding mode $r \in \{\uparrow, \downarrow\}$. They take as input sets of environments ($\mathcal{E} = \mathcal{V} \to \text{Val}$) mapping variables to values (which are either integers or dates), and produce values (resp. environments).

$$\mathbb{E}_r[\![expr]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\text{Val}) \qquad \qquad \mathbb{S}_r[\![stmt]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$$

We define in Fig. 7 the concrete semantics evaluating expressions and statements over both rounding modes, written respectively $\mathbb{E}_{\updownarrow}[\![expr]\!]$ and $\mathbb{S}_{\updownarrow}[\![stmt]\!]$. We do not delve into the details of product programs, which are defined in the work of Delmas et al. [21]. In this semantics, the state is now duplicated: $\mathcal{D} = \mathcal{E} \times \mathcal{E}$. We ensure that random operations return the same value in both rounding modes, to avoid spurious desynchronizations. The sync predicate returns true if and only if the expression evaluates to the same values in both rounding modes, capturing the rounding-insensitivity of the contained expression. We use it in the programs we analyze to target the expressions we want to check, as we have already seen in Fig. 4. The evaluation of other expressions is performed pointwise on both rounding modes, and similarly for the assignments.

Definition 4. *An expression e is* rounding-insensitive *in a state d if and only if $\mathbb{E}_{\updownarrow}[\![sync(e)]\!](\{d\}) = \{(true, true)\}$. This property is encoded in programs by the statement* `assert(sync(e))`.

$$\mathbb{E}_{\updownarrow}[\![expr]\!] : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\text{Val}^2)$$
$$\mathbb{E}_{\updownarrow}[\![\text{random\_date()}]\!](D) = \{(d, d) \mid d \in \mathbb{Z}^3, valid(d)\}$$
$$\mathbb{E}_{\updownarrow}[\![\text{sync}(e)]\!](D) = \bigcup_{(\rho_\uparrow, \rho_\downarrow) \in \mathcal{D}} \{(b_u == b_d, b_u == b_d) \mid (b_u, b_d) = \mathbb{E}_{\updownarrow}[\![e]\!](\rho_\uparrow, \rho_\downarrow)\}$$
$$\mathbb{E}_{\updownarrow}[\![expr]\!](D) = \bigcup_{(\rho_\uparrow, \rho_\downarrow) \in \mathcal{D}} \{(v_\uparrow, v_\downarrow) \mid v_\uparrow = \mathbb{E}_\uparrow[\![e]\!]\rho_\uparrow, v_\downarrow = \mathbb{E}_\downarrow[\![e]\!]\rho_\downarrow)\}$$
$$\mathbb{S}_{\updownarrow}[\![stmt]\!] : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$$
$$\mathbb{S}_{\updownarrow}[\![v = e]\!](D) = \bigcup_{(\rho_\uparrow, \rho_\downarrow) \in D} \{(\mathbb{S}_\uparrow[\![v = v_\uparrow]\!]\rho_\uparrow, \mathbb{S}_\downarrow[\![v = v_\downarrow]\!]\rho_\downarrow), (v_\uparrow, v_\downarrow) \in \mathbb{E}_{\updownarrow}[\![e]\!]\{\rho_\uparrow, \rho_\downarrow\}\}$$

Fig. 7. Concrete semantics over double evaluation of rounding modes

The abstract semantics mimics the concrete behavior, but works on a single abstract state instead of a set of concrete double states. The double state is represented by duplicating variables according to their rounding mode in the numerical abstract domain. A variable `x` is thus written $\uparrow$`x` (resp. $\downarrow$`x`) to represent the variable when the upper (resp. lower) rounding mode is used. This duplication is performed in a shallow fashion to improve usability: when performing an assignment `x = e`, if `e` evaluates into the same value in both rounding modes, the variable `x` will not be duplicated into the numerical abstract domain.

Example 5 (Rounding-sensitivity of the comparison). Back to our running example, we have shown so far how the YMD domain analyzes the program when rounding up (Ex. 4). Continuing with the same relational abstract domain, we show part of the abstract state in the partition focusing on rounding to Feb. 28 of a non-leap year in Eq. (1). In the rounding mode down, `intermediate` rounds to Feb. 28, and thus `limit` rounds down to Feb. 1st.

$$\begin{aligned}
&\text{day}(\texttt{current}) \in [1, 31], \text{month}(\texttt{current}) \in [1, 12], \text{year}(\texttt{current}) \in [-\infty, +\infty]\\
&\text{day}(\texttt{birthday}) = 29, \text{month}(\texttt{birthday}) = \texttt{Feb}, \text{year}(\texttt{birthday}) \equiv_4 0\\
&\uparrow\text{day}(\texttt{intermediate}) = 1, \uparrow\text{month}(\texttt{intermediate}) = \texttt{Mar}\\
&\downarrow\text{day}(\texttt{intermediate}) = 28, \downarrow\text{month}(\texttt{intermediate}) = \texttt{Feb}\\
&\downarrow\text{year}(\texttt{intermediate}) = \uparrow\text{year}(\texttt{intermediate}) = \text{year}(\texttt{birthday}) + 2\\
&\uparrow\text{day}(\texttt{limit}) = 1, \uparrow\text{month}(\texttt{limit}) = \texttt{Mar} \downarrow\text{day}(\texttt{limit}) = 1, \downarrow\text{month}(\texttt{limit}) = \texttt{Feb}\\
&\downarrow\text{year}(\texttt{limit}) = \uparrow\text{year}(\texttt{limit}) = \text{year}(\texttt{birthday}) + 2
\end{aligned} \tag{1}$$

We exhibit an abstract state where we cannot prove that the expression `current < limit` is rounding-insensitive. The static analysis will consider all cases in the comparison and the evaluation in both rounding modes. For the sake of presentation here, we only highlight one case. The date comparison operator between `current` and the rounded up version of `limit` yields a partition where the years are the same and the month number is less. This partition refines the abstract state above with the following constraints:

$$\text{year}(\texttt{current}) = \uparrow\text{year}(\texttt{limit}) \land \uparrow\text{month}(\texttt{limit}) < \text{month}(\texttt{current}) = \texttt{Mar} \tag{2}$$

Let us now consider the case where the comparison with the rounded down version of `limit` does not hold, when the years and months are the same but the days are not. We get the following additional constraints:

$$\begin{aligned}&\text{year}(\texttt{current}) =_{\downarrow} \text{year}(\texttt{limit}) \wedge \text{month}(\texttt{current}) =_{\downarrow} \text{month}(\texttt{limit}) = \texttt{Feb} \wedge \\ &\text{day}(\texttt{current}) \geq_{\downarrow} \text{day}(\texttt{limit}) = 1\end{aligned} \tag{3}$$

Combining the constraints from Eqs. (2) and (3) on the abstract state from Eq. (1) gives the following result on `current`:

$$\text{year}(\texttt{current}) = \text{year}(\texttt{birthday}) + 2 \wedge \text{month}(\texttt{current}) = \texttt{Feb} \tag{4}$$

To summarize, our analysis has been unable to prove the rounding-insensitivity of the expression `current < limit`, in particular in the case of the abstract state presented in Eq. (1), refined with constraints from Eqs. (2) and (3). Thanks to partitioning and relational abstract domains, we know that the proof fails when `birthday` is a Feb. 29th (of a year y which is divisible by 4, a sound but not complete way to express it is leap). In that case, `intermediate` will either be Feb. 28th or March 1st of $y + 2$. This entails that `limit` will either be Feb. 1st or March 1st of $y + 2$. In the cases where `current` is a day of February of $y + 2$ (Eq. (4)), the comparison will effectively be rounding-sensitive.

The original program did not contain any constraints on `birthday` or `current`. Note that if we add in the program that the day of `birthday` is less than 28, our analysis is able to automatically prove the program to be rounding-insensitive.

4.4 Implementation

We implemented our approach in the Mopsa static analysis platform [28, 29]. Mopsa is able to analyze C, Python and multilanguage Python/C programs [40, 41, 44], to prove the absence of runtime errors, and to perform portability analysis of C programs [21]. We modified the front-end of a toy imperative language also available in Mopsa to analyze programs performing date arithmetic. We chose to extend this language for our analysis as we do not require advanced features from C nor Python. Thanks to Mopsa's modular architecture, we have been able to reuse iterators for intraprocedural analysis with little code changes. The configuration used by Mopsa for our analysis is illustrated in Fig. 8. The "D.bidates" domain corresponds to the abstract domain and transfer functions described in Sec. 4.3. The "U.ymd" domain is the YMD domain (Sec. 4.1). The last part enclosed in a gray box corresponds to the numerical abstract domain on top of which the YMD domain was built (Sec. 4.2).

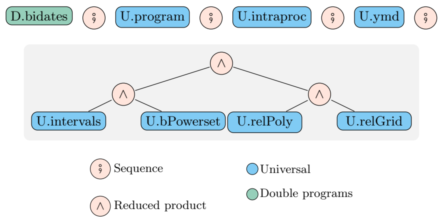


Fig. 8. Date analysis configuration

```
5: assert(sync(current < limit));
               ^^^^^^^^^^^^^^^
Desynchronization detected: (current < limit). Hints:
↑month(limit) = 3, ↑day(limit) = 1, ↓month(limit) = 2, ↓day(limit) = 1,
↑month(intermediate) = 3, ↑day(intermediate) = 1,
↓month(intermediate) = 2, ↓day(intermediate) = 28,
month(birthday) = 2, day(birthday) = 29, month(current) = 2, day(current) = [1,29],
year(birthday) =[4] 0, year(current) = ↑year(intermediate) = ↑year(limit)
= ↓year(intermediate) = ↓year(limit) = year(birthday) + 2
```

Fig. 9. Mopsa's output on the running example

4.5 Generating counter-example hints

We have extended our implementation to provide counter-examples hints when a synchronization assertion cannot be proved safe. Given our usecase, it is paramount to provide meaningful feedback to users translating law articles into Catala code so they understand why their date computations might be ambiguous (Sec. 5). These hints are precise constraints on the considered program that may lead an expression to be rounding-sensitive. They are especially helpful to provide more precise date ranges for unconstrained dates that may affect rounding sensitivity. As our approach is incomplete, these hints may be spurious; we however did not encounter this issue in our case study on Catala programs.

This generation of counter-example hints is atypical for static analyses by abstract interpretation. This approach is permitted here by a simplified setting (variables are assigned once, and the abstract state is partitioned to ensure a high precision) and the use of powerful relational abstract domains. In a general setting with multiple variable assignments, joins and widenings, most approaches need to perform backward analyses [1, 38, 49].

This generation of hints works in two steps: it first starts by heuristically selecting the best partition of the abstract state. The YMD domain may partition the abstract state in order to keep the best precision. Our heuristic selects the partition with the highest number of desynchronized variables (meaning there has been significant roundings), and the highest number of auxiliary variables for days and months which are constants. The second step of the hint generation extracts the relevant constraints from the considered abstract state. This extraction starts by collecting all date variables defined in the program. For these variables, we evaluate the auxiliary day, month and year variables into intervals, and keep only intervals providing meaningful information (i.e., intervals strictly included in $[1, 31]$ for day variables, strictly included in $[1, 12]$ for month variables, and bounded intervals for year variables). We then project the relational abstract domain onto the set of auxiliary variables where no meaningful intervals has been extracted to provide linear relations for those. We show in Fig. 9 the exact, unedited output of the hints generated by Mopsa in the case of our running example and highlight their readability. They correspond exactly to the constraints previously described in Ex. 5.

5 Case Study: Application to Catala

This section highlights how the results and methods established in the previous section can be applied in the setting of legal expert systems, and more specifically within Catala [34], a recent domain-specific programming language designed to be understandable by lawyers and close to the structure of legal texts, with formal semantics that clearly define its behavior to reduce discrepancies between legal texts and their implementation.

We start by describing rulings and implementations of the law where precise and well-defined date arithmetic is paramount to ensure expected results. Then, we describe how Catala's implementation of date rounding has recently evolved: from the issues we noticed in Catala's previous off-the-shelf implementation, to the port to our date calculation library and the introduction of a function-local rounding definition when legal references or interpretations are known, reducing the number of cases where the rounding mode is unspecified. We finish by explaining the latest implemented feature, which allows the Catala compiler to extract date computations and relies on Mopsa to (dis)prove rounding-insensitivity.

5.1 Date arithmetic and the law

Critical software relying on date computations is commonly used by companies or government agencies to automatically enforce legal dispositions, e.g., to check if an application has been filed within the correct time period, to compute age-related conditions, or to aggregate periods between dates and compare the result to a fixed duration for eligibility calculation.

In all these cases, there can be heavy financial and legal consequences when a date computation goes wrong or is subject to diverging interpretation. In the case Bowles v. Russell, 551 U.S. 205 (2007) cited by Bailey [7], the court gave Bowles a 17-days notice to file an appeal but this notice was incorrectly computed from Rule 4(a)(6) and paragraph 2107(c), as it should have been 14 days. When Bowles filed his appeal on the 17th day, the court system dismissed the appeal on the basis that Bowles should have filed on the 14th day and not trust the notice the court gave him earlier. In more mundane cases, an incorrect date computation can deprive someone of their social benefits, or impose a higher late fee than what should be.

These doubts about date computation in software applying the law are all the more concerning that previous research in code open-sourced by French government agencies did not show a great deal of transparency or trustful practices on that particular matter. For instance, the custom programming language M, used by the French tax authority to compute income tax [35], encodes dates as mere floating-point numbers where the date is just a decimal number in the format DDMMYYYY. The French unemployment agency, whose IT system is mostly implemented in Java, uses a custom date library for its computation (`fr.unedic.util.temps.Damj`) but its implementation is omitted from their only open-source release [47].

5.2 Catala's policy about date rounding

Recently, the Catala project [24, 34] has aimed to bring more accountability and transparency to programs computing taxes or social benefits inside government agencies. The Catala language is specifically designed to allow the easy translation of computational law into code; in particular, it is based on prioritized default logic [10], which enables programmers to closely follow the base case/exception pattern that permeates the law. To increase confidence and explainability in its programs, Catala also comes with a formal semantics which is formalized in the F* proof assistant. These formal semantics mostly focus on Catala's default calculus, the encoding of prioritized default logic as a programming language, and do not specify all Catala expressions, including date computations.

Initially, the semantics of the date computations was defined by the behavior of the `calendar` OCaml library [50] used inside its interpreter. However, this library relies on the POSIX behavior which is not always monotonic and may appear quirky (for instance, it computes `Jan 31st + 1 month` as `March 3rd` for non-leap years) despite its very complete set of features. These unusual behaviors prompted a deeper investigation about the corner cases of date computations and led to the implementation of the library presented in this paper. While now integrated in the Catala interpreter, our library is standalone, and freely available with an open-source license. As the Catala compiler is implemented in OCaml, so is our library[5], currently packaged with opam; however, by relying on our semantics, its implementation is straightforward. We do not foresee any difficulty porting it to other languages, and plan to do so to support more of the Catala backends, including Python and JavaScript.

The default behavior of our date computation library inside the Catala interpreter is to raise a runtime exception whenever a date rounding is needed during a computation. This choice of behavior has been made conservatively because the decision to round up or down date computations in software enforcing legal rules is itself a legal rule that has to be specified, as we described in the introduction of this paper. To avoid runtime exceptions, rounding rules can be specified at the scope level (a precise definition of Catala's scopes is outside the range of this paper, but it can be considered as a sort of function in Catala) and should be justified, for example by a legal reference or interpretation.

We applied this methodology to fix the code of the biggest Catala program so far, which computes the French housing benefits [32]. Articles L822-4, R823-4 of the construction and housing Code, as well as article L512-3 of the social security Code, all feature a comparison of the age of the user to an age constant. However, as the input to the Catala program is not the age of the user but their birth date, we know such a comparison can be ambiguous if the user was born on February 29th on a leap year and if the current date is March 1st. In those situations, we took the decision to round up the date addition, as shown in Fig. 10, with the `date rounding increasing` mention. We are currently trying to contact the relevant government agencies operating the system for clarifications about how this issue should be handled.

[5] Our F* formalization can be extracted to executable but non-idiomatic OCaml code. In practice, we thus manually reimplement our library in OCaml to use features such as named arguments or exceptions to provide a more idiomatic API.

```
declaration scope CheckingAgeInferiorEqual:
  input birth_date content date
  input current_date content date
  input target_age content duration # always a number of years
  output age_is_inferior_or_equal_target content boolean

scope CheckingAgeInferiorEqual:
  definition age_is_inferior_or_equal_target equals
    birth_date + target_age <= current_date
  date rounding increasing
```

Fig. 10. Catala code for checking the age of the user is lower than a constant

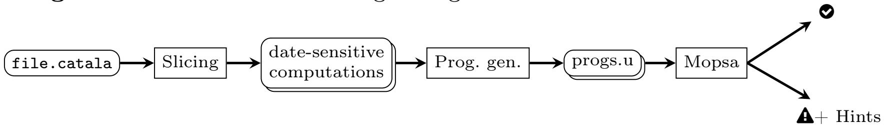


Fig. 11. Catala date ambiguity analysis pipeline

To best benefit the recipient and be in line with the general principle underpinning legal interpretations of social security law in France, a better solution would be to perform the computation twice, by rounding up and down, and select the outcome most favorable to the user in case of disagreement. The flexibility offered by our library allows us to do that, and we intend to explore this avenue in future work. Being able to control precisely where the rounding is done and how is key for developers and maintainers of such programs, as they are responsible for the legal effect of the program itself [22].

5.3 Detecting potentially ambiguous computations

Choosing the rounding mode for each date computation allows us to precisely control the outcome of ambiguous computations. However, given the pervasiveness of such computations in legal texts, it is also extremely tedious, and figuring out the cases where an ambiguous computation could happen is complex. For these reasons, we expect some developers to delay this step and wait for incidents to figure out the policy of the institution operating the program on the matter. But figuring out this policy might itself be tricky because of the automation frontier [33] strictly separating the developers from the decision-makers in charge of legal policy decisions.

To help developers reach out to the legal services of their institution with concrete examples of where things can go wrong before production incidents, we integrated the semantics and abstract domains presented in this paper inside the ongoing initiative to provide a proof platform for Catala programs [20]. By connecting the Catala compiler to the Mopsa static analyzer, we are able to check whether a date computation can be ambiguous in the context of the program, and often exhibit a counter-example if it is the case. We present in Fig. 11 our analysis pipeline. It consists of three main phases: program slicing, verification condition crafting, and analysis – which may generate counter-examples.

First, we scan the Catala program in one of its intermediate representation and look for Catala expressions susceptible of raising a runtime exception because of an ambiguous date computation. We use classic techniques of program slicing for this step, selecting only the target sub-expression and then adding the definitions of variables used in that sub-expression recursively to extract a small, self-contained program with sufficient information to be analyzed. This will simplify the counter-example hint generation of Mopsa, which outputs constraints on variables rather than subexpressions of a computation.

Second, we augment the sliced program with the assertions and other information about its variables that are declared in the original Catala program to further constrain the search space. So far, our analysis is intraprocedural, but we are planning to implement an inlining pass to make it inter-procedural. We then translate the sliced program to Mopsa's toy language (using the `.u` extension), which can then be fed to the static analyzer.

Finally, we run Mopsa on the generated program. As we have mentioned in Sec. 4.5, Mopsa is able to exhibit potential counter-examples hints. While these hints are approximate due to incompleteness of the analysis, they are often sufficient to yield real, actionable counter-examples on the Catala programs that we analyzed. We extract relevant intervals and linear constraints and display them to the user, in the format illustrated by Fig. 9. While the intervals and constraints presented are descriptive, and sufficient for a programmer to identify concrete counter-examples, they can however be difficult to grasp for non-experts. Formatting these constraints in a more readable format is an interesting question, requiring further interaction with lawyers; we leave it as future work.

The implementation of housing benefits in Catala currently consists of about 20,000 lines (including the text of the law directly specifying it) that were written prior to this work. While automatically analyzing this implementation using our verification pipeline, we found issues in two date computations (one of them being our running example). In both cases, Mopsa was able to provide actionable counter-example hints. Several other computations were age computations, which are now handled by a custom scope with a legally interpreted date rounding mode, as shown in Fig. 10. Finally, remaining computations rely on durations defined outside of the analyzed scope, which requires an inter-scope analysis in Catala, which is being implemented. In the meantime, we performed a manual duration extraction in these cases and detected 16 new unsafe (rounding-sensitive) date comparisons, which are real issues. In all cases, the provided counter-example hints are actionable. In 10 cases, the issues can only happen with a current date before 2023. By constraining the year to be greater or equal to 2023, these 10 cases are proved safe. All date arithmetic programs we have currently extracted or written are small and analyzed within three seconds.

As the number of Catala programs grows, we hope to apply our analyzer at a larger scale, possibly suggesting future avenues for improvement.

6 Related Work

We start by surveying the behavior of mainstream implementations of date arithmetic. We created a suite of litmus tests involving date-duration additions, and the expected result depending on the rounding mode. We wrote test drivers for each library, running those tests to decide which rounding mode applies.

The `java.time` library [43] provides a `LocalDate` class for dates and a `Period` class to express durations. In our tests, the addition is performed by rounding down. This behavior is explicitly described in the documentation [26]. To the best of our knowledge, there is no option to use another rounding mode, or fail during ambiguous computations. In the Python standard library, the `datetime` module [46] provides a `date` class and `timedelta` to express durations. However these durations cannot be defined in terms of months, but only in terms of days. A third party library called `dateutil` [45] provides a replacement feature, `relativedelta`, able to express durations in months and years. This library seems widely used, as it ranks within the top 20 most downloaded Python packages. On our tests, this library rounds down. This seems to be confirmed by the documentation stating that "adding one month will never cross the month boundary." Similarly to Java, this rounding behavior is not configurable. The `boost` C++ [9] and the `luxon` [31] JavaScript libraries exhibit similar behaviors.

The `coreutils` implementation of date arithmetic follows a different principle, which is not expressible in our semantics. When adding months, this implementation first computes an adjusted date which might not be valid. This adjusted date d_a is then normalized using POSIX's `mktime` function. For example, adding one month to 2023-03-31 yields adjusted date 2023-04-31, which does not exist and is normalized into 2023-05-01. In this case, the behavior is the same as the upper rounding. There are however cases where its behavior differs: adding one month to 2023-01-31 yields adjusted date 2023-02-31, which is normalized into 2023-03-03. This behavior breaks monotonicity of the addition in the date argument (2023-02-01 + 1 MONTH is 2023-03-01). In ambiguous computations, the debug mode of the `date` utility outputs a warning with the following message "when adding relative months/years, it is recommended to specify the 15th of the months" – which is a sufficient condition to avoid any ambiguity. This semantics is also followed by the `calendar` [50] library of OCaml.

We finish this survey with the case of spreadsheet editors (such as Google Sheets), and highlight an inconsistent behavior we have found in them. The `EDATE` function adds a given number of months to a date. In our experiments, this function silently rounds down. As such, adding 18 years (that is, 216 months) to 2004-02-29 yields the date 2022-02-28. These spreadsheets applications also offer the `DATEDIF` function, which can compute the duration in years between two dates. In that case, `DATEDIF`(2004-02-29, 2022-02-28) yields 17 years (18 years are reached when the second date is 2022-03-01). This behavior is inconsistent with `EDATE`. Cheng and Rival [11] focus on performing a type analysis of spreadsheet applications, given that a runtime type casting may silently happen and provide unwanted results (similarly to what JavaScript does). This analysis supports a variety of types, including dates, but as it focuses on type information there is no mention of the value semantics of operations on dates.

The book of Reingold and Dershowitz [48] can be seen as the hacker's delight of calendar computations, with many efficient formulas for day additions, and a wide range of different calendars being presented. Their work does not mention nor address the issue of month addition, and potential date rounding, which is at the core of our work. Although we have not needed it for now, we could leverage their approach to optimize the recursive computations of our library. Similarly, ISO 8601 defines the representation of dates in the Gregorian calendar, but does not address date-duration additions with years or months.

The Formal Vindications start-up developed a mechanized, formally verified implementation of a time management library [2, 3] in Coq, computing over dates and time, including specific technical points (timezones, leap seconds). Their duration of a month is defined as 30 days. Some recent changes allow to round down dates. A similar effort was developed in Lean 4 by Bailey [6], but this library only supports the addition of days to a date. As a reminder, the Catala project currently targets laws that do not need to go beyond the precision of a day in terms of time management. Formal Vindications developed a formally verified, high-precision tachograph software for enforcing truck drivers scheduling laws [19].

We finish this related work by highlighting similarities between floating-point and date arithmetic. Floating-point arithmetic is more complex and widely used, but both settings have rounding operators with different modes available. This similarity has guided us in our search for properties that hold and counter-examples presented in Sec. 3. The static analysis to prove non-ambiguity of date computations presented in Sec. 4 can be seen as the abstract execution of the computation under both rounding modes, to compare results. To the best of our knowledge, no such static analysis for floating-point programs try to bound the difference in computations between two rounding modes. Tools such as Daisy [8, 18], Fluctuat [23] and FPTaylor [51] usually aim at upper-bounding errors between ideal computations over reals and a machine computation using floating-point.

7 Conclusion and Future Work

Legal expert systems rely on date computations, which are ambiguously defined in some corner cases. There are different ways of solving these ambiguities through different rounding operators, where no operator prevails over the others. We have thus defined semantics for date computations, taking into account these ambiguities to either raise errors, or round the result (either up or down). This semantics has been implemented into a publicly available OCaml library. We have studied this semantics and have formally proved several properties they satisfy, and exhibited counter-examples to usual properties they do not satisfy. We have defined and implemented an analysis that is able to prove an expression to be *rounding-insensitive* in a given program. This analysis relies on partitioning and relational abstract domains to maintain the best possible precision, and can generate understandable counter-examples hints. Both our library and the rounding-sensitivity analysis have been integrated within the Catala language – which focuses on implementing computational laws. Through our analysis, we found rounding-sensitivity issues in the implementation of the French housing benefits in Catala. We surveyed the behavior of mainstream date arithmetic libraries, and developed litmus tests that can be used to test new libraries.

There are limitations to our static analysis: its soundness has not been proved mechanically, but the proofs simply lift theorems that have been formally verified. The current analyzed language is a core imperative language which was sufficient for our case studies. Having an inter-scope analysis within the Catala to Mopsa translation would improve our precision in the case study. We plan to craft human-readable error messages from Mopsa's output. We believe the relevant constraints are already properly extracted by Mopsa and the rest of the work consists in engineering, in order to inverse the translation from Catala date computations to Mopsa programs.

In spite of these limitations, we believe this paper to be a crucial step into clarifying and improving the robustness of many computer programs implementing "business logic", often overlooked by formal methods. The widespread use of date arithmetic in programs used by companies or government agencies to operate massive financial transfer should have prompted a formal analysis of date rounding a long time ago, but the existing literature only indicates a recent interest from the formal methods community on the matter.

This work was triggered by the problems we found during interdisciplinary investigations about French housing benefits using the Catala programming language. From these investigations surfaced the need for various formal analysis, which we have thus started integrating into the programming language. We hope to further develop the integration of static analysis into the Catala proof platform, thus benefiting both legal and computer science users by including formal methods advances into development processes of Catala programs.

Artifact Availability Statement. All our development is under open-source licenses, public or in the process of being upstreamed into a public development. To foster reproducibility of our results, we provide an artefact [39] containing the formal proofs written in F*, our date calculation library, and our ambiguity detection analysis as well as supporting evidence of our case study.

Acknowledgements. We thank the anonymous reviewers for their constructive feedback and support of our work. We are obliged to Abdelraouf Ouadjaout for making his implementation of partitioning within Mopsa available to us. We are grateful to David Delmas for the discussions around double semantics, Liane Huttner & Sarah Lawsky for the interesting discussions around the properties this work targets, and Louis Gesbert for his technical help around the Catala compiler. We appreciated the many discussions and valuable feedback about this work we got from the whole Catala team.

Bibliography

[1] Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS, Lecture Notes in Computer Science, vol. 7214, pp. 157–172, Springer (2012)

[2] 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: CPP, pp. 2–13, ACM (2024)

[3] de Almeida Borges, A., Buñuel, Q.C., Rodriguez, J.C., Bedmar, M.G., Reyes, E.H.: Formal vindication time library. https://gitlab.com/formalv/formalv/-/tree/dev/theories/time (2023)

[4] Arora, C.M.: What is the concept of "month" while computing limitation period under the custom act? https://itatonline.org/digest/articles/what-is-the-concept-of-month-while-computing-limitation-period-under-the-custom-act/ (2020)

[5] Bagnara, R., Dobson, K.L., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: LOPSTR, Lecture Notes in Computer Science, vol. 4407, pp. 219–235, Springer (2006)

[6] Bailey, C.: A date and time library for lean 4, implementing the proleptic gregorian calendar. https://github.com/ammkrn/timelib (2023)

[7] Bailey, C.: Research keynote at the Programming Languages and the Law workshop (POPL). https://www.youtube.com/watch?v=_0ZycH8NB4s (2023)

[8] Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A verified certificate checker for finite-precision error bounds in coq and HOL4. In: FMCAD, pp. 1–10, IEEE (2018)

[9] Boost contributors: The Boost c++ libraries. https://www.boost.org/ (2023)

[10] Brewka, G., Eiter, T.: Prioritizing default logic. In: Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel, pp. 27–45, Springer (2000)

[11] Cheng, T., Rival, X.: Static analysis of spreadsheet applications for type-unsafe operations detection. In: ESOP, Lecture Notes in Computer Science, vol. 9032, pp. 26–52, Springer (2015)

[12] Chevalier, M., Feret, J.: Sharing ghost variables in a collection of abstract domains. In: VMCAI, Lecture Notes in Computer Science, vol. 11990, pp. 158–179, Springer (2020)

[13] Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the International Conference on Functional Programming (ICFP) (2000)

[14] Cornell Law School: Rule 6. computing and extending time; time for motion papers. https://www.law.cornell.edu/rules/frcp/rule_6 (2016)

[15] Council of the European Communities: Regulation (eec, euratom) no 1182/71 of the council of 3 june 1971. https://eur-lex.europa.eu/

LexUriServ/LexUriServ.do?uri=CELEX%3A31971R1182%3AEN%3AHTML (1971)

[16] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (1977)

[17] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96, ACM Press (1978)

[18] Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: TACAS (1), Lecture Notes in Computer Science, vol. 10805, pp. 270–287, Springer (2018)

[19] de Almeida Borges, A., Conejero Rodríguez, J.J., Fernández-Duque, D., González Bedmar, M., Joosten, J.J.: To drive or not to drive: A logical and computational analysis of european transport regulations. Information and Computation (2021), 26th International Symposium on Temporal Representation and Reasoning

[20] Delaët, A., Merigoux, D., Fromherz, A.: Turning Catala into a Proof Platform for the Law. In: Workshop on Programming Languages and the Law at POPL 2022, Philadelphia, United States (Jan 2022)

[21] Delmas, D., Ouadjaout, A., Miné, A.: Static analysis of endian portability by abstract interpretation. In: SAS, Lecture Notes in Computer Science, vol. 12913, pp. 102–123, Springer (2021)

[22] Diver, L., McBride, P., Medvedeva, M., Banerjee, A., D'hondt, E., Duarte, T., Dushi, D., Gori, G., van den Hoven, E., Meessen, P., Hildebrandt, M.: Typology of legal technologies (2022)

[23] Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI, Lecture Notes in Computer Science, vol. 6538, pp. 232–247, Springer (2011)

[24] Huttner, L., Merigoux, D.: Catala: Moving Towards the Future of Legal Expert Systems. Artificial Intelligence and Law (Aug 2022)

[25] Internal Revenue Service: Exclusion of gain from sale of principal residence. https://www.law.cornell.edu/uscode/text/26/121 (2017)

[26] Java™ Platform Standard Ed. 8: Class LocalDate. https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#plusMonths-long- (2023)

[27] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, Lecture Notes in Computer Science, vol. 5643, pp. 661–667, Springer (2009)

[28] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, Lecture Notes in Computer Science, vol. 12031, pp. 1–18, Springer (2019)

[29] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: The MOPSA static analyzer (2023), URL https://gitlab.com/mopsa/mopsa-analyzer/

[30] Legifrance: Article 641 - code de procédure civile. https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000006411002 (1976)

[31] Luxon contributors: Luxon — a library for working with dates and times in javascript. `https://github.com/moment/luxon` (2023)

[32] Merigoux, D.: Experience report: implementing a real-world, medium-sized program derived from a legislative specification. In: Workshop on Programming Languages and the Law 2023, POPL 2023, Boston (MA), United States (Jan 2023)

[33] Merigoux, D., Alauzen, M., Slimani, L.: Rules, Computation and Politics: Scrutinizing Unnoticed Programming Choices in French Housing Benefits. Journal of Cross-disciplinary Research in Computational Law **1**(3) (2023), (forthcoming)

[34] Merigoux, D., Chataing, N., Protzenko, J.: Catala: a programming language for the law. In: Proceedings of the International Conference on Functional Programming (ICFP) (2021)

[35] Merigoux, D., Monat, R., Protzenko, J.: A modern compiler for the french tax code. p. 71–82, CC 2021, Association for Computing Machinery (2021)

[36] Microsoft: Filetime (minwinbase.h). `https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-filetime` (2021)

[37] Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI, Lecture Notes in Computer Science, vol. 3855, pp. 348–363, Springer (2006)

[38] Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program. **93**, 154–182 (2014)

[39] Monat, R., Fromherz, A., Merigoux, D.: Formalizing Date Arithmetic and Statically Detecting Ambiguities for the Law (Artifact) (Jan 2024), `https://doi.org/10.5281/zenodo.10460049`

[40] Monat, R., Ouadjaout, A., Miné, A.: Static type analysis by abstract interpretation of python programs. In: ECOOP, LIPIcs, vol. 166, pp. 17:1–17:29, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)

[41] Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native C extensions. In: SAS, Lecture Notes in Computer Science, vol. 12913, pp. 323–345, Springer (2021)

[42] Monin, J.: Louvois, le logiciel qui a mis l'armée à terre. `https://www.radiofrance.fr/franceinter/podcasts/secrets-d-info/louvois-le-logiciel-qui-a-mis-l-armee-a-terre-8752880` (2018)

[43] Oracle: Package java.time. `https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html` (2023)

[44] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: SAS, Lecture Notes in Computer Science, vol. 12389, pp. 223–247, Springer (2020)

[45] Python-dateutil contributors: python-dateutil — useful extensions to the standard python datetime features. `https://github.com/dateutil/dateutil` (2023)

[46] Python Software Foundation: datetime — basic date and time types. `https://docs.python.org/3/library/datetime.html` (2023)

[47] Pôle Emploi: Calcul de l'allocation d'aide au retour à l'emploi (are). https://www.pole-emploi.org/opendata/calcul-de-lallocation-daide-au-r.html (2018)
[48] Reingold, E.M., Dershowitz, N.: Calendrical Calculations: The Ultimate Edition. Cambridge University Press (2018)
[49] Rival, X.: Understanding the origin of alarms in astrée. In: SAS, Lecture Notes in Computer Science, vol. 3672, pp. 303–319, Springer (2005)
[50] Signoles, J.: The calendar ocaml library (2011), URL https://github.com/ocaml-community/calendar
[51] Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. **41**(1), 2:1–2:39 (2019)
[52] Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (2016)
[53] The Open Group: time. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/functions/time.html (2017)
[54] The QCheck contributors: Q-check: quickchecking library for ocaml. https://github.com/c-cube/qcheck (2023)
[55] Torny, D.: L'administration sanitaire entre contraintes techniques et contraintes juridiques. L'exemple des maladies émergentes. Revue générale de droit médical (6) (2005), URL https://shs.hal.science/halshs-01249084

Author Index